Escuela
Politécnica
Superior

Arquitectura Domótica de bajo coste

Grado en Ingeniería Informática

Trabajo Fin de Grado

Autor:

Jaime Moreno Cantó

Tutor/es:

Otto Colomina Pardo

Universitat d'Alacant
Universidad de Alicante

AGRADECIMIENTOS

A mi pareja (Arancha Ferrero) por ayudarme siempre en temas de estilo y de JavaScript.

A mi amigo de la infancia Javier Pérez por aconsejarme y ayudarme en todo el tema de componentes eléctricos.

A mi otro amigo, también de la infancia, Adolfo Folgueral por realizarme un plano de la maqueta en 3D.

A mi tío Philip Hamer por ayudarme a montar la maqueta con la que puedo defender mi trabajo.

A los "Pistoleros del Eclipse" (Robert Escaplez, Iván Mora, Álvaro Muñoz) por darme siempre opinión y consejos de todo lo que he realizado durante el trabajo.

A mi tutor (Otto Colomina) por orientarme en este camino.

Y a mis padres (Antonio Moreno, Isabel Cantó) por haber confiado en mi proyecto y haberlo "financiado".

ÍNDICE

INTRODUCCIÓN

En el documento presente se relatan todos los pasos seguidos para la elaboración de una arquitectura domótica teniendo en cuenta aspectos de usabilidad de cara al usuario, seguridad y fiabilidad. Para acompañar y ver una puesta en marcha de dicha arquitectura se han confeccionado a su vez dos clientes con ayuda de las librerías JavaScript más usadas hoy en día, que más tarde veremos en detalle.

El objetivo del proyecto es tener de nuestra mano un sistema completo de domótica de bajo coste y, sobretodo, completamente acoplable a inmuebles ya construidos sin ningún tipo de base domótica, es decir, instalable en cualquier tipo de circuito eléctrico convencional de cualquier domicilio.

El objetivo a largo plazo del proyecto es, además de poder controlar remotamente nuestro inmueble, minimizar los costes de energía de la misma. El poder establecer rutinas, por ejemplo el enchufado y apagado de los botelleros de un bar y cafeteras, supone el ahorro de una cantidad considerable de dinero al cabo de un año.

Para ello se ha tenido que realizar una labor de ingeniería para minimizar al máximo los costes para así poder competir con otros sistemas de domótica ya existentes en este gran mercado que es el de las casas inteligentes.

Por supuesto si algo nos han enseñado en el grado es que los proyectos software nunca mueren, siempre están en constante evolución. Un proyecto software muere cuando muere la finalidad para la que está hecho. Es por eso que, aunque haya una fecha de entrega, nunca les podré decir que mi proyecto esté "acabado". Me gustaría sinceramente que esto no quedara sólo en un trabajo de grado si no que avanzara hasta algo que pudiera originar un beneficio a alguien en un futuro.

Sin más, les dejo los diferentes enlaces a repositorios de GitHub donde están estructurados los diferentes componentes que precisa el proyecto que nos ocupa:

- API REST en NodeJS
- Cliente en Python para la comunicación API-Rasperry Pi
- Cliente Web en ReactJS
- Cliente móvil multiplataforma en React-Native

ESTADO DEL ARTE

A continuación se indicarán y detallarán diferentes herramientas existentes en el mercado actual con un parecido similar a la propuesta de este trabajo, realizando una comparación con el resultado final del mismo.

Dispositivos IoT

Existen infinidad de dispositivos inteligentes que podemos conectar a nuestros hogares: bombillas, enchufes, aire acondicionado, sensores de movimiento, etc. La gran mayoría pueden ser controlados desde, bien dispositivos móviles u ordenadores o bien desde algún dispositivo facilitado por el propio fabricante, como puede ser una tablet.

Todos estos dispositivos tienen la ventaja del "plug-&-play", por lo que como usuarios únicamente nos debemos preocupar de descargarnos la interfaz software que maneja el dispositivo. No obstante esto conlleva a que tengamos múltiples interfaces para domotizar enteramente nuestro domicilio, ya que por cada dispositivo que adquiramos normalmente habremos de descargar la interfaz indicada por el fabricante para poder controlarlo.

Además, si intentamos domotizar todo el domicilio con este tipo de dispositivos la cantidad de dinero que deberíamos invertir sería similar al que deberíamos invertir si tuviéramos un sistema de domótica a medida para nuestro inmueble. Por ejemplo, teniendo en cuenta que una bombilla preparada para IoT Philips Hue E27 de 9W cuesta unos 23€ en tienda española, deberíamos emplear unas 17 bombillas para un piso de unos cien metros cuadrados. Contando con el kit de inicio del propio fabricante para poder manejar las bombillas (que cuenta con dos de ellas) se nos ajustaría el precio en unos 95€ en tienda española. Si calculamos obtenemos que, únicamente para la iluminación básica de techo, hemos empleado una cuantía de 440€ aproximadamente, mientras que con nuestro sistema utilizaríamos la instalación eléctrica existente en el hogar.

Home Assistant

Enlazando con lo anterior, "Home Assistant" es un proyecto gratuito y de código abierto para permitirnos gestionar nuestros dispositivos IoT de domótica en una sola aplicación. El proyecto está escrito en Python3 y diseñado para ser ejecutado en una Rasbperry Pi, haciendo uso de IFTTT para la comunicación con los diferentes dispositivos y pudiendo integrarse con una gran variedad de APIs como HUE, Nest y WeMo.

IFTTT es una plataforma de software que conecta apps, dispositivos y servicios de diferentes desarrolladores y fabricantes con el carácter de activar uno o varias automatizaciones relacionadas con las tres entidades comentadas anteriormente. Se basa en el pensamiento "If This Then That". Actualmente hay más de 54 millones de applets de IFTTT. No está pensado para aplicaciones puramente domóticas si no que su pretexto es de un ambiente más general, por ejemplo para apuntar una entrada en el log cuando se recibe una llamada por teléfono.

Volviendo a la aplicación, para manejarla se debe hacer una configuración que para el usuario común puede ser tediosa y poco intuitiva ya que requiere de la escritura de múltiples ficheros de configuración de la aplicación: red y máscara de red de la red a utilizar, scripts de automatización de luces, etc. Todo esto requiere un conocimiento básico de programación, capacidad que muchos usuarios finales no tienen.

Además el software opera a nivel de dispositivo mientras que el proyecto presentado aquí opera a nivel de la instalación eléctrica del inmueble. Esto quiere decir que puede que no todos los dispositivos se relacionen bien con el software o bien que no se puedan realizar diferentes acciones por las limitaciones del dispositivo. Además al operar directamente con la red eléctrica nos ahorramos el tener que reemplazar o suplementar todos los dispositivos eléctricos de la casa que queramos domotizar.

Google Home

Se trata de un altavoz inteligente con las mismas funciones de Google Assistant, pero con la ventaja de poder usarlo como reproductor de audio o como controlador de otros dispositivos de nuestra casa.

Google Home precisa, a la vez que todos los dispositivos que pueda manejar, una conexión a la red Wi-Fi de nuestra casa, por lo que es raro el inmueble en el que no se pueda instalar con relativa facilidad. Con la aplicación Google Home de Android y cinco minutos de actualización y puesta en marcha, el dispositivo estará perfectamente configurado para operar sin mayores problemas.

Aquí tenemos un listado de las cosas que, por el momento, Google Home puede realizar:

- Responder preguntas directas, por ejemplo qué tiempo hace o cuándo juega nuestro equipo favorito. Lógicamente buscando en su motor, Google.
- Responder preguntas indirectas, gracias al machine learning. Por ejemplo cuando tenemos hambre o estamos aburridos.
- Asistente personal: alarmas, recordatorios, etc
- Hogar inteligente y conectado. Se puede conectar con Netflix, Spotify y cualquier dispositivo inteligente, siempre y cuando tenga un enchufe inteligente (véase el punto de "Dispositivos IoT").
- Entretenimiento. Básicamente puede contar chistes malísimos y enseñarnos algunas recetas básicas de cocina, como una tortilla de patatas.

No obstante, Google Home sigue necesitando de terceros para funcionar como es debido. Precisa que los dispositivos estén domotizados de primeras, ya que únicamente los maneja, tal y como hace la aplicación de Home Assistant, comentada anteriormente. Además, para usar plataformas como Netflix y Spotify, Google Home necesita que la televisión pertinente esté conectada a un Chromecast de Google. Esto, al haber muchas televisiones que ya de por sí son inteligentes, muchas veces es innecesario y, por tanto, Google Home no nos podría hacer la función.

El sistema presentado, como ya se ha comentado antes, tiene la ventaja de operar directamente sobre red eléctrica. Pese a que actualmente no se tengan tantas

funcionalidades como Google Home, no se precisa de plataformas de terceros para poder funcionar.

Amazon Alexa

Con Amazon Alexa no nos vamos a demorar mucho más que con Google Home, puesto que se trata exactamente del mismo dispositivo con las mismas funcionalidades, pero con ciertas diferencias que lo deja en una peor posición, al menos para el territorio español.

Mientras que Google Home y, por supuesto, nuestro sistema ofrecen compatibilidad perfecta en español, Alexa no está ahora mismo soportado en nuestro idioma ni nuestro territorio. De hecho lo más próximo que podemos hacer para configurarlo es dejarlo en inglés de Reino Unido, sin posibilidad de instalar una app oficial en español.

Por esto, Alexa nos da las temperaturas en Farenheit en vez de Celsius, es incapaz de indicarnos cualquier localización cercana al depender de su aplicación oficial, y además las alarmas son ineficaces al contar con una zona horaria que no es la misma que la del territorio español.

Amazon sin embargo ofrece una documentación oficial que permite a terceros desarrollar para la plataforma de Alexa. Esto nos puede ser útil en un futuro por integrar nuestra plataforma con Alexa, bien para combinar funcionalidades o bien para el manejo de nuestro sistema por voz, dejando la implementación de dicha funcionalidad de lado.

Domótica a medida (KNX)

Existen empresas que realizan una instalación domótica en el inmueble correspondiente, teniendo que realizar un desarrollo en exclusiva para la configuración de dicho inmueble.

Estos procesos, como cualquier otro software desarrollado a medida, conlleva unos costos de proyecto que asume directamente el cliente, ya que es un producto que únicamente va a tener un beneficiario (o unos cuantos) en concreto, sin posibilidad de implantación en otros entornos.

Normalmente, esta implantación domótica a medida se realiza con el estándar KNX. La asociación KNX representa a los fabricantes de dispositivos para las aplicaciones basadas en KNX.

Lo que nuestro sistema palia, es que es una arquitectura perfectamente instalable en cualquier domicilio/negocio, pero sin la necesidad de estar a medida. Lo que ha conseguido es que sea un sistema perfectamente configurable para cualquier tipo de inmueble que posea una red eléctrica de corriente alterna.

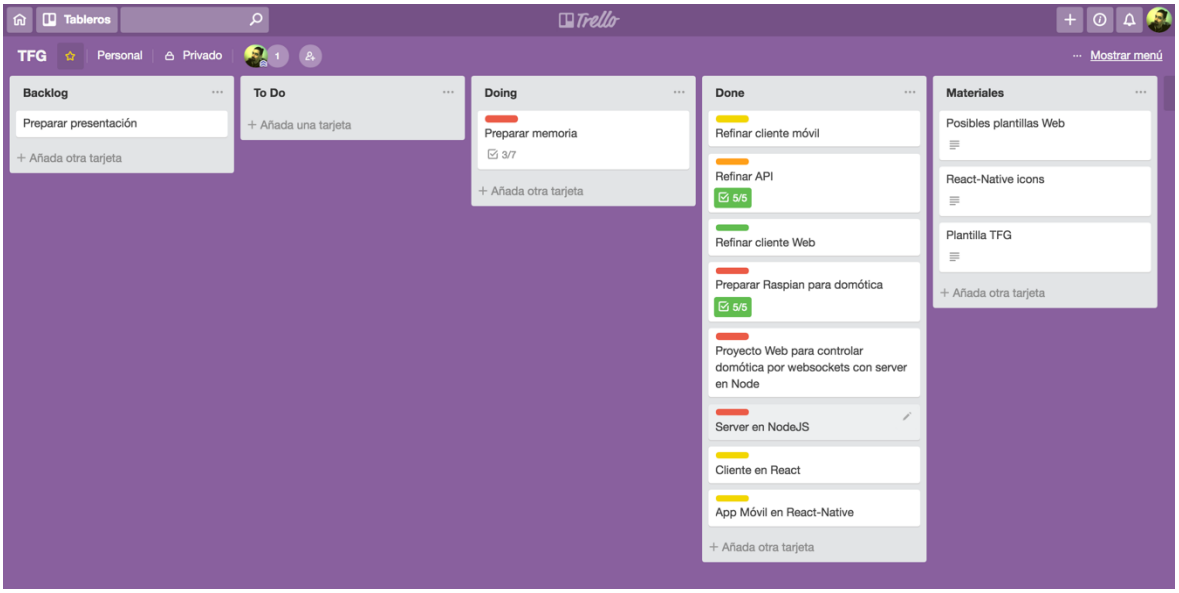
Es por esto que al usuario final le va a suponer un menor costo, ya que al ser una arquitectura desarrollada para poder ser utilizada en casi cualquier inmueble, el coste del servicio que se de no recae únicamente en uno, o en unos pocos usuarios finales.

METODOLOGÍAS DE DESARROLLO

Últimamente estamos viendo un gran auge de las metodologías de desarrollo ágiles que, personalmente hablando, comparto después de haber trabajado en el mundo laboral de manera tradicional y de manera ágil. Creo que una gran cantidad de gente se está dando cuenta o se ha dado cuenta ya de que las metodologías antiguas no tenían afinidad con el mundo del desarrollo del software, por lo que poco a poco estamos cambiando nuestra manera de hacer las cosas en nuestro sector, y creo que para bien.

Todo este párrafo viene a la idea de que se nos ha enfocado (por lo menos en el itinerario de Ingeniería del Software) a la idea de que hemos de trabajar en equipo, que es lo que vamos a hacer en el mundo laboral. Y efectivamente, así es. Es por ello que se ha intentado seguir una metodología, como he comentado, ágil. No obstante no se ha pretendido en ningún momento emplear puramente ninguna de sus especialidades (Scrum, Kanban, etc) ya que muchas de ellas precisan de un equipo mínimo de tres o cuatro personas para llevarse a cabo. En lugar de ello se han cogido ideas interesantes de cada metodología.

Para la organización de tareas se ha escogido un tablero Kanban con la herramienta de Trello. Con Trello se ha conseguido de alguna manera especificar una serie de iteraciones con las tareas más prioritarias y que más valor nos han dado a lo largo del proyecto:



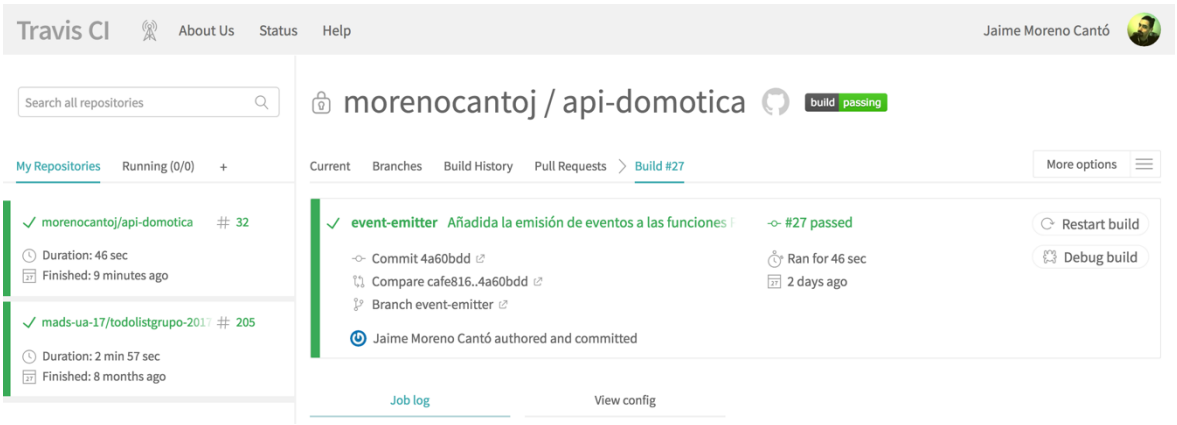
Como podemos ver en la imagen, tenemos cuatro columnas exceptuando la columna de materiales, que está ahí por razones ajenas a la metología. Como vemos tenemos la columna de "Backlog", que nos indica cuantas tareas nos quedan para "finalizar" el proyecto de final de grado. A la derecha tenemos las columnas de "To Do" y "Doing", que representan por una parte las tareas escogidas para la iteración y, por otra parte, la o las tareas que estamos realizando en el momento. La última columna, como es obvio, representa todas las tareas que hemos finalizado. Los colores además nos marcan una importancia gradual de la tarea según el color que lo acompañe, siendo el verde el color menos importante hasta el rojo, que indica prioridad máxima.

En lo referente al código se ha usado un sistema de control de versiones bastante extendido en todo el planeta: Git, hosteado en GitHub (explicado más adelante). La metodología de desarrollo en cuanto al código ha sido bastante fácil: rama principal master y sub-ramas para desarrollar nuevas funcionalidades, refinar existentes o simplemente para temas de refactorización. Al finalizar cada funcionalidad se abre un pull-request de la sub-rama a master y si todo está correcto se mergea, integrando todo el código nuevo a la rama principal, evitando en la mayor medida los conflictos que se puedan ocasionar.

Con este sistema también vemos un claro reflejo de nuestro tablero Kanban con nuestro repositorio, ya que cada tarea o subtarea se re reflejada en una rama aparte dentro de Git.

Para el desarrollo del software, como vemos, se ha utilizado una metodología de Integración Continua. No obstante con el desarrollo de la API REST se ha ido un paso más allá usando Travis CI, una herramienta de Integración Continua que, como veremos más adelante, nos abstrae de casi todo el trabajo que necesita esta metodología.

Como hemos visto en la imagen anterior del Pull-Request, cada vez que hacemos un commit en el proyecto de la API se pasan una serie de tests que nos indican si la build de ese commit ha pasado todos los tests o ha fallado.



Con este sistema hemos conseguido, por ejemplo, no mergear nuestra sub-rama con la rama principal si hay cualquier error que no hemos contemplado al pasar algún test en producción. Primero se pasan los tests en local, luego en la build de Travis, y luego lo volvemos a probar una vez se ha mergeado todo el código para quedarnos completamente seguros.

Siguiendo el proceso de Integración Continua en la API, se han configurado tres entornos según la situación en la que estemos en el proceso de desarrollo: development, test y producción. Cuando estamos en la primera, los tests se ejecutan en torno a la base de datos local de nuestra máquina, mientras que cuando estamos en la fase de test (build en Travis), se crea una base de datos específica para esa build con un script guardado previamente, por lo que estamos usando otra BBDD diferente a la vez. Por último, cuando estamos en el entorno de producción se apunta directamente a una base de datos externa a nuestro sistema, con valores totalmente enfocados a producción, es decir, valores reales.

Es por esto último que se ha configurado la base de datos de producción para que haga copias de seguridad de sí misma cada cierto tiempo, procurándonos siempre un respaldo en caso de que tengamos cualquier fallo en producción o situaciones graves como ataques y hackeos. Una práctica tradicional en servidores compartidos pero que nunca está de más para preservar la seguridad en el entorno real.

HERRAMIENTAS Y TECNOLOGÍAS

A continuación revisaremos las diferentes herramientas que se han empleado en el la arquitectura y proceso de desarrollo del proyecto, indicando las aclaraciones y justificaciones pertinentes al "por qué" del uso de dichas tecnologías.

Dividiremos las diferentes herramientas en varios apartados:

- **Tecnologías para el lado del servidor**: NodeJS, MySQL, Heroku
- **Tecnologías para el lado del cliente de domótica**: Raspberry Pi 3, Python3
- **Tecnologías para el lado del cliente**: React, React-Native
- **Otras tecnologías**: Protocolo Websockets, Travis CI, Trello, Git

NodeJS

Node.js es un entorno de ejecución para JavaScript construido con el motor de JavaScript V8 de Chrome. Es mayormente utilizado para construir APIs REST, como es nuestro caso, en el lado del servidor con la ayuda de librerías como Express, pero puede ser utilizado para generar contenido web dinámico, entre otras cosas.

La mayor ventaja de la que goza NodeJS es a la hora de manejar peticiones con el cliente. Mientras que los lenguajes convencionales como PHP necesitan esperar hasta que la acción que les requiere acabe para procesar la siguiente petición, NodeJS se ocupa de todo de forma asíncrona, por lo que no necesita esperar a que la tarea termine para escuchar la siguiente petición. Simplemente, cuando la tarea es realizada, realiza las acciones que deba. NodeJS en resumen elimina la espera y simplemente se dedica a atender la siguiente petición.

Además NodeJS no deja de ser un lenguaje JavaScript, por lo que podemos usar módulos (similares a las librerías JavaScript) para facilitarnos la tarea a la hora del desarrollo, contando así con múltiples módulos publicados en la red instalables de forma rápida y sencilla con herramientas como npm o yarn. Además podemos usar nuestros propios módulos exportándolos desde diferentes ficheros JavaScript indicando su ruta a la hora de importarlo, por lo que nos brinda de herramientas muy potentes para tener nuestro código limpio y bien separado en zonas distintas.

NodeJS funciona únicamente en monohilo, con la ventaja comentada de la programación asíncrona. Es por esto que es en lenguaje que es más eficiente en términos de memoria que otros lenguajes que cumplan la misma función, como por ejemplo el anteriormente indicado PHP.

MySQL

MySQL es un motor de base de datos relacional "open source" utilizado como respaldo de datos en organizaciones tan potentes como Facebook, Google, Adobe o Alcatel. Además MySQL se usa en gran parte de las SaaS ya que es un motor de base de datos potente, seguro y fácil de usar.

Como ventajas, MySQL ha sido diseñado y optimizado específicamente para aplicaciones Web. Además MySQL es altamente fácil de escalar, por lo que supone un ahorro monetario importante a la hora de hacer cualquier tipo de cambio dentro de la estructura de la base de datos, a la par de que es sencillo de mantener por los administradores de base de datos. Además MySQL-Enterprise es un servicio bastante menos costoso que otros motores relacionales como Microsoft SQL Server Enterprise Edition, llegando a tener un ahorro de 700000$ en tres años de vida de un proyecto.

Pero no únicamente contamos con el factor económico. Las constantes actualizaciones de MySQL se hacen notar. Comparando MySQL 5.5 con MySQL 5.1 es hasta un 360% más rápido en instrucciones R/W y hasta un 200% más rápido en operaciones de solo lectura. Todo esto en servidor Linux ya que en servidor Windows es un 1500% y un 500% de ganancia respectivamente.

Pero la mayor razón de elegir MySQL no es más que por su simpleza y su facilidad de uso. La mayoría de administradores de base de datos tienen un alto nivel de manejo con MySQL, por lo que resulta fácil replicar bases de datos en producción en servidores y computadores de pruebas: máquinas virtuales, demonios, exportaciones completas de SQLs, etc.

Por último, al tener un esquema de base de datos no tan complejo no se ha visto necesario el emplear recursos en transformar el esquema para el uso de base de datos no relacional, ya que no mejoraría en gran medida el rendimiento. Además podemos externalizar el servicio de MySQL en una entidad completamente ajena a nuestro proceso (como veremos a continuación), lo que nos libra del mantenimiento de la misma y de la gestión de errores que puedan ocurrir.

Heroku

Heroku es una plataforma en la nube que permite a las compañías construir, entregar, monitorizar y escalar aplicaciones. Según la propia descripción de su página web, son "el camino más rápido para ir desde la idea la URL, pasando por alto todos esos dolores de cabeza que producen las infraestructuras".

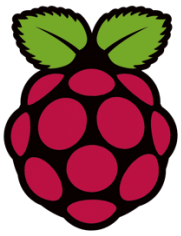
Y, efectivamente, la plataforma de Heroku nos facilita muchísimo las cosas a la hora de publicar una aplicación en NodeJS por ejemplo, como es en este caso. Heroku además tiene soporte para otros lenguajes que son los más utilizados en las aplicaciones modernas: Ruby, Java, Scala, PHP, entre otros.

Heroku nos brinda además "addons" que facilitan la integración de nuestra aplicación con diferentes servicios, por ejemplo bases de datos Postgresql o MySQL. En nuestro caso hemos delegado nuestro respaldo de datos a un addon de Heroku de MySQL.

Además de todo esto Heroku ofrece ayuda inmensa en torno al desarrollo de integración continua. Nuestro proyecto en Heroku es configurable para integrarlo junto con nuestro repositorio en GitHub, por lo que podemos configurar también que el proyecto se republique cuando cierta rama de nuestro repositorio, pudiendo así sacar a la luz versiones nuevas de nuestra aplicación y controlarlas de manera sencilla. Además de que en cada integración podemos a su vez ejecutar los tests que hayamos realizado en la API.

Para finalizar, se ha elegido Heroku también porque ofrece una alta personalización sin la localización de nuestro código en un servidor compartido. Esto nos permite acceso a terminal por SSL y una mayor seguridad al recaer la mayoría de su peso en nosotros, además de que no tenemos que hacer configuraciones complejas de acceso con archivos htaccess, por ejemplo. Además Heroku nos ofrece un servicio gratuito con certificado SSL que, una vez más, nos abstrae de otra configuración costosa tanto en tiempo como monetaria.

Raspberry Pi 3



La plataforma Raspberry Pi no es más que un minicomputador todo en uno desarrollado por la Fundación Raspberry Pi en Cambridge, que cabe en la palma de la mano. Es un computador completo que soporta varios componentes necesarios en un ordenador común y es capaz de comportarse como tal. El modelo que nos atañe posee 1GB de memoria RAM, varios puertos USB, una salida de vídeo y audio HDMI, tecnología Bluetooth, adaptador de red inalámbrico y, lo más importante,

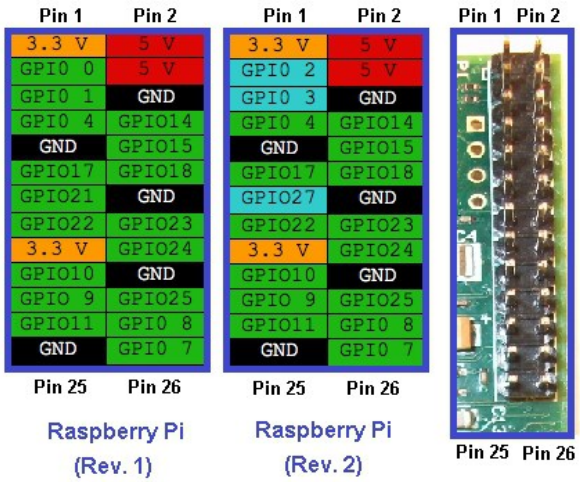
una serie de puertos GPIO que son los fundamentales para el fin de este proyecto, que más tarde explicaremos en detalle.

Todo esto alimentado con un simple cargador micro USB, como el de los móviles de hoy en día y funcionando sobre un sistema operativo Linux/Unix para procesadores ARM, como puede ser un Debian adaptado específicamente para funcionar en esta herramienta.

Se ha optado por esta plataforma en vez de otra similar, como podría ser Arduino, por la libertad de configuración. Pese a que la interfaz y uso de Arduino a priori pueda ser más sencilla, y consta también de los puertos GPIO anteriormente indicados, Raspberry Pi nos provee de un sistema completo a un coste similar, lo que nos permite una configuración más extensa y detallada a nuestro gusto. Arduino por su parte solo permite ejecutar un solo programa por las especificaciones de su hardware, por lo que se nos limitaría muchísimo a la hora de realizar múltiples tareas no relacionadas con una porción de código: servicios, tareas cron, etc.

La arquitectura Raspberry Pi ha sido y es actualmente una de las grandes plataformas donde se desarrollan proyectos de diferente índole: domótica, inteligencia artificial, BI, reconocimiento facial y de voz, etc. Es por esto que hay infinidad de tutoriales para el manejo de dispositivos con el uso de librerías que actúan directamente sobre los puertos GPIO de la placa.

Para finalizar con esta arquitectura explicaremos un poco en detalle los pines de Propósito General Input/Output (GPIO).



La Raspberry Pi tiene 26 pines (puertos BCM), dependiendo del modelo y versión, organizados en dos filas. Cada uno de esos pines o bien tiene una función específica o bien es de propósito general (GPIO). Por ejemplo la segunda revisión de la Raspberry dispone de dos pines de 5V, dos de 3,3V (señal lógica del procesador), ocho pines GPIO, cinco de toma a tierra y unos pocos más que no nos interesan en nuestra labor. Con ellos podemos controlar circuitos electrónicos, por ejemplo para mover motores o controlar el encendido y apagado de relés que actuarán como interruptores para circuitos de 220V.

Los pines GPIO son configurables y controlables desde el intérprete bash de la terminal, o bien desde lenguajes de programación como C o Python, gracias a las librerías mencionadas anteriormente. Es por eso que definitivamente se ha elegido esta arquitectura para el desarrollo de una arquitectura domótica de bajo coste.

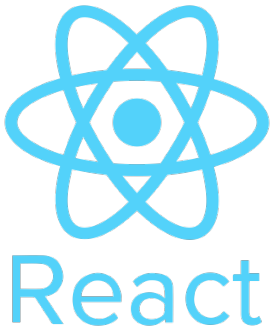
Python 3

Python es un lenguaje dinámico multiplataforma y multiparadigma, por lo cual permite crear aplicaciones en una amplia variedad de sistemas operativos. Su licencia de código abierto permite su utilización en distintos contextos sin necesidad de pagar por ello. Su objetivo es automatizar procesos para ahorrar tanto complicaciones como tiempo.

La elección de Python para la comunicación entre la API en NodeJS y la Raspberry Pi se basa en su sencillez, legibilidad y exactitud en la sintaxis, ya que Python es un lenguaje elegante y minimalista. No obstante Python, al ser un lenguaje interpretado, es más lento que Java o C++, por ejemplo. Sin embargo no nos afecta en gran medida puesto que la única labor que debemos hacer con la Raspberry Pi es abrir y cerrar puertos GPIO y comunicarnos mediante websockets (más adelante hablaremos de ello) con la API REST subida a Heroku.

Todo esto, junto a las librerías GPIO y de websockets que hay, hacen de Python una excelente herramienta para una integración de forma sencilla entre nuestra arquitectura Raspberry Pi, nuestro circuito integrado con ésta y nuestra API RESTful.

React + React-Native



ReactJS es básicamente una librería open-source de JavaScript desarrollada por Facebook utilizada para construir interfaces de usuario específicamente para Single Page Applications (SPA).

React además nos permite crear componentes de interfaz reusables. Además permite a los desarrolladores crear aplicaciones web de tamaño considerables, que pueden cambiar datos, sin recargar la página, similar a cuando usamos la librería AJAX de JavaScript para hacer peticiones a APIs, por ejemplo.

React-Native por su parte es una librería nativa anunciada por Facebook en 2015 que provee una arquitectura para exportar aplicaciones nativas para plataformas móviles como iOS, Android o Windows Phone.

Hay dos motivos fundamentales por los que se ha elegido la tecnología de React para la elaboración de los clientes que interactúan con la API en NodeJS. Uno de ellos es el poder disfrutar de la experiencia JavaScript: aplicaciones dinámicas que cambian sus componentes gráficos en tiempo real, como si de una aplicación de escritorio se tratara. Si bien podría haberse hecho con JavaScript más el plugin de jQuery React nos ayuda y nos simplifica bastante el proceso. El otro motivo por el que no utilizar otras herramientas como Angular es que React y React-Native al fin y al cabo son lo mismo, por lo que todo el código relacionado con la integración con API e integración de estado es, literalmente, copiar y pegar de un repositorio a otro.

Gracias a esto ha sido posible la reutilización de una gran parte de código, que además es de la parte más compleja de ambas aplicaciones. Es por eso que sin duda utilizar React ha sido una decisión acertada en cuanto a la interacción entre el usuario final y el sistema de domótica.

Protocolo Websockets

Websockets es una tecnología basada en el protocolo ws bajo el protocolo TCP, que hace posible establecer una conexión continua full-duplex entre cliente y servidor, por lo que, a resumidas cuentas, el servidor también puede "hablar" también al cliente y mandarle un mensaje el cualquier momento. Es el sistema utilizado en un gran porcentaje de aplicaciones de chat entre usuarios, como puede ser WhatsApp.

Para iniciar una conexión con el protocolo WS primero el cliente le pide al servidor iniciar dicha conexión (handshake). Una vez el servidor confirma se deja de usar http (usado para establecer dicha primera conexión) y se empieza a usar WS hasta que uno de los dos extremos decide dejar de comunicarse, cerrando así la conexión. Además, al igual que existe un protocolo http seguro (https), existe también la versión segura de WS: WSS.

Se ha usado esta tecnología para la comunicación entre el cliente de la Raspberry Pi en Python 3 y la API en NodeJS para evitar el "long-polling" entre ambos, como ya veremos en apartados siguientes. Websockets permite la comunicación por eventos (encendido de una luz, cambios de temperatura…), ahorrándonos así infinidad de llamadas innecesarias que necesitaríamos con la comunicación por http con la API.

Además existen multitud de módulos y librerías para la integración con NodeJS y Python 3, por lo que se nos facilita en gran medida el desarrollo para estos lenguajes. Además Heroku ofrece soporte para dicho protocolo, lo que no nos supone tampoco un problema a la hora de poner toda la arquitectura en producción.

Travis CI

Travis CI es una herramienta de integración continua gratuita para todos los proyectos open-source alojados en GitHub. Con tan solo un archivo de configuración alojado en la raíz del proyecto podemos automatizar todo el proceso de integración continua para que Travis lo haga por nosotros: tests, builds en ramas principales o en pull-requests, deployments en Heroku… etc.

Se ha elegido Travis por varias razones. La primera de ellas es que se puede configurar el archivo de Travis para poder subir directamente el proyecto a Heroku si la build pasa todos los tests correctamente, por lo que nos ahorra el "trabajo" de tener que acordarnos de subir cierta build a producción. Con esto, Travis se encarga directamente de todo poniendo directamente la rama desde la que queremos que se haga la subida.

La segunda razón es porque Travis tiene una afinidad muy buena con GitHub. Se pueden configurar múltiples marcadores en los archivos de texto markdown de nuestro repositorio en GitHub para que nos muestre si las builds de Travis pasan los tests, se han cancelado o directamente ha fallado la build. Además en el archivo de configuración de Travis se puede configurar para que se hagan deploys o builds según ramas específicas o pull-requests.

Gracias a esto Travis CI nos ahorra un enorme trabajo, sobretodo en términos de testeo y tiempo a la hora de la gestión de builds para ponerlas en producción.

Trello

Trello es una plataforma en la nube que simula un tablero Kanban. En dicha plataforma podemos crear listas que representan columnas con un título en nuestro tablero Kanban, pudiendo rellenarlas de tarjetas indicando la tarea a realizar, prioridad, si tiene subtareas, adjuntar archivos, etc.

Se ha elegido Trello puesto que no se ha necesitado de una herramienta más compleja para gestión de proyectos como puede ser Jira. Al final el proyecto consta de una persona, e incluso si no era necesario un programa para la organización de las diferentes tareas y requisitos (se podría haber hecho simplemente en papel), sí que ha ayudado a la visualización de lo que había por hacer, qué había hecho y cuál es el punto en el que se había dejado.

Además se ha aprovechado para, por ejemplo, en la fase de recolección de información e ideas para poder organizar todos los links en varias tarjetas de una columna de Trello.

Git

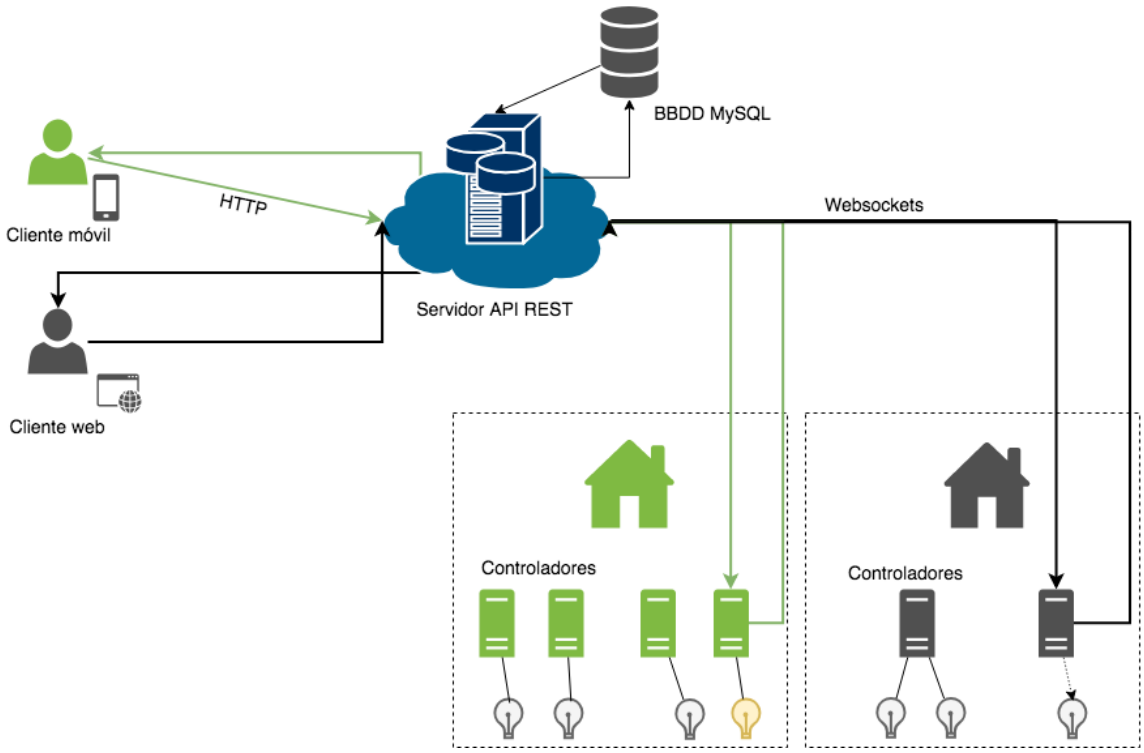
Git es, sin lugar a dudas, el SCV más utilizado a nivel global. Git fue creado por Linus Torvalds pensando en la eficiencia y la confiabilidad del mantenimiento de versiones de aplicaciones. Git nos proporciona las herramientas para desarrollar un trabajo en equipo gracias a su enfoque distribuido.

Precisamente por esto se ha elegido Git, además de ser una herramienta archiconocida y que la tenemos bastante más trabajada que otras herramientas distribuidas como Mercurial. Al ser un SCV distribuido no estamos copiando únicamente la última versión del repositorio, sino que estamos copiando todo el

repositorio completo, por lo que si se perdiera el servidor del repositorio fuente, cualquier cliente que se lo haya descargado puede copiarlo en otro servidor y así tenerlo disponible de nuevo, cosa que es imposible en los SCV centralizados, puesto que se copia la última versión del fichero con el que estamos trabajando.

Como servidor para el alojamiento del código se ha elegido la plataforma de GitHub ya que, como hemos visto en la sección anterior y al explicar Travis CI, la integración con esta herramienta nos facilita enormemente la visión, sobretodo a nivel de equipo, de cómo está la situación en el proyecto, tanto de manera global como específica al entrar en un Pull-Request o rama en específico.

ARQUITECTURA DEL SISTEMA



En la imagen presentada vemos lo que representa la arquitectura del sistema de domótica desarrollado en este proyecto.

La idea es disponer en cada inmueble una serie de controladores (equipos Raspberry Pi 3) a la que se puedan conectar dispositivos de diversa índole de la instalación eléctrica: bombillas de luz, enchufes, electrodomésticos, climatización…

Dichos controladores se comunican con el servidor mediante websockets, identificándose en la primera conexión con el servidor, guardándose en una estructura mapa cada conexión con los controladores.

Los controladores están soportados por una arquitectura ARM Raspberry Pi 3, funcionando sobre un sistema operativo Debian adaptado para la arquitectura de la Raspberry Pi. Cada controlador se comunica con el servidor en la nube gracias a la tarjeta de red inalámbrica de la que dispone, teniendo salida a internet. No necesitamos IPs fijas en los inmuebles ya que son los propios controladores los que inician la conexión con el servidor. No obstante sí se necesita una conexión estable a Internet de, al menos, 600 Kbits/s de subida y 2 Mbits/s de bajada para que la comunicación entre Raspberry-servidor no suponga una carga representativa para otras tareas de los usuarios con la red.

El servidor actúa como API RESTful para los clientes web y móvil, mientras que se comunica, como ya se ha comentado, por websockets, teniendo ambas partes en el mismo elemento. Dicho servidor es accesible desde los controladores gracias a que está subido a la plataforma de Heroku, lo que le da acceso a un dominio (DNS) accesible sin necesidad de conocer IPs. El respaldo de datos está garantizado por un servicio externo de base de datos MySQL, dispuesto por la plataforma de Heroku.

El modo de funcionamiento es sencillo. Los usuarios (cliente web y móvil) interactúan con la API, mientras que ésta, si es necesario actualizar un dispositivo físico, crearlo o borrarlo, interactúa por websockets con el controlador asignado correspondiente, efectuando la orden que se indica por el servidor. Los cambios de datos se efectúan a nivel de base de datos, por lo que las funcionalidades están claramente diferenciadas. Una vez que la petición es finalizada, se devuelve una respuesta al usuario, que según el cliente, se puede representar de maneras diferentes, ya que esta se encuentra en un cuerpo JSON.

Veamos un poco más en detalle los diferentes componentes que definen la arquitectura presentada.

Controlador Raspberry Pi

El controlador Raspberry Pi es el encargado de procesar las peticiones que vienen desde el servidor API REST, el cual se comunica con éste mediante el protocolo Websockets (explicado anteriormente).

Entre sus funciones se encuentran, a día de hoy:

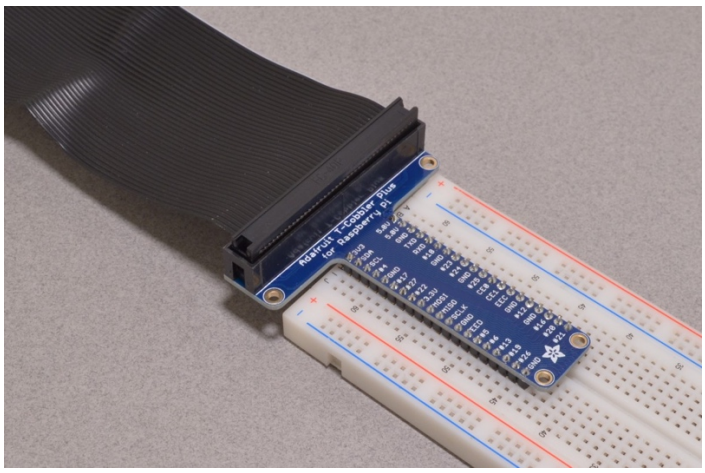
- Encendido y apagado de puertos GPIO
- Programación de encendido y apagado de puertos GPIO

Lo importante del controlador es que el controlador no sabe realmente lo que está haciendo. Realmente el controlador es una máquina tonta que realiza las acciones que le manda la API.

Para poner en marcha, por ejemplo, un controlador con un pequeño LED, debemos realizar unos pequeños pasos. No obstante, antes debemos realizar una pequeña instalación:

- En primer lugar se ha de conectar la Raspberry a un T-Cobbler, que no es más que una extensión de los pines BCM de los que dispone la Raspberry
- En segundo lugar, debemos conectar dicho extensor a una tabla de conexiones sin soldaduras, de tal forma que nos resulte cómodo operar con los puertos BCM

El resultado debería ser algo similar a esto:

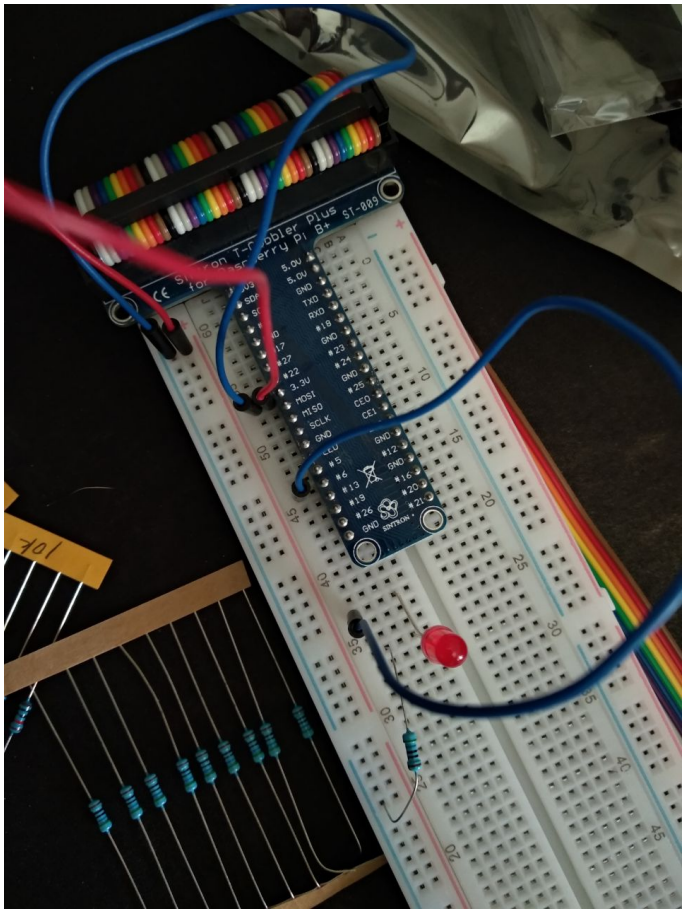


El cable negro junto con el pequeño dispositivo son el T-Cobbler (extensor de pines), que está conectado a una tabla de conexiones.

Realizado esto, simplemente nos queda conectar nuestro pequeño LED. Para realizar dicha conexión debemos realizar los siguientes pasos:

- El pin largo del LED lo conectamos a una extensión de un puerto GPIO, que es el necesario para activar y desactivar el LED. Recordemos que al extender un puerto con un cable Jumper, todas las clavijas libres de la línea donde esté conectado dicho cable estarán conectadas al mismo puerto
- El pin corto lo conectaremos a una resistencia o directamente a toma de tierra (puerto GND). Si usamos una resistencia ésta la deberemos conectar a un puerto de +/- 3.3V, usando de nuevo un cable Jumper.

El resultado debería ser el siguiente, si usamos una resistencia:



Si nos fijamos en la imagen, el cable azul conectado al puerto GPIO número 6 está conectado a un extremo del LED. Simplemente añadiendo un dispositivo con este puerto en nuestro cliente web ya podríamos manejarlo sin mayor problema.

Servidor API REST

El servidor API REST es el cerebro de toda la arquitectura. Es el encargado de procesar las peticiones de los clientes web y móvil y, si fuera el caso, enviar las instrucciones al controlador Raspberry Pi. Se encarga además de interactuar con una base de datos MySQL para el almacenado de datos: usuarios, inmuebles, controladores, dispostivos, estado de dispositivos, programaciones… Toda la información pasa primero por la API antes de ser enviada a cualquiera de los extremos de la comunicación.

Sus funcionalidades son, representadas por rutas de la API:

- Registro de un nuevo usuario (POST)
- Inicio de sesión de un usuario existente, devolviendo un token de sesión único (POST)
- Ver toda la información sobre los inmuebles de un usuario (GET)
- Ver toda la información sobre los controladores de una casa (GET)
- Ver toda la información sobre un controlador en específico, con todos los dispositivos conectados a ese controlador (GET)
- Crear un nuevo dispositivo en un controlador específico (POST)
- Editar un dispositivo existente de climatización para editar la temperatura (PUT)
- Editar un dispositivo existente de luz para apagar o encender el dispositivo (PUT)
- Crear una programación de fecha y hora para un dispositivo, ya sea luz o climatización (POST)
- Borrar un dispositivo de un controlador en específico (DELETE)
- Ver todas las programaciones pendientes de un controlador (GET)
- Ver todos los eventos ocurridos en un controlador (GET)

Todas las rutas POST/PUT/DELETE (exceptuando el registro y login por razones obvias) están protegidas con un middleware de autenticación, al que se le debe pasar el token obtenido previamente en el inicio de sesión.

El dominio de la API se encuentra en https://mighty-reef-55430.herokuapp.com/, por lo que contamos con seguridad SSL también.

Clientes Web y Móvil

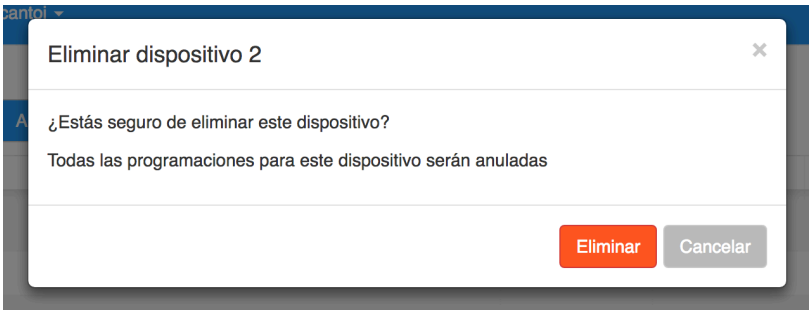
Los clientes web y móvil son los encargados de operar con la API de una forma cómoda para el usuario. Pese a que son compatibles entre sí, también tienen algunas diferencias entre ellos. El cliente web está más pensado para labores de administración, mientras que el cliente móvil está pensado para el uso cotidiano del día a día en la domótica, ya que resulta más incómodo realizar operaciones de administración con un Smartphone.

Las funciones comunes entre los dos clientes son:

- Iniciar sesión y cerrar sesión como usuario
- Ver los inmuebles de un usuario logueado
- Ver los controladores de un inmueble específico
- Ver el controlador en detalle, junto con todos sus dispositivos
- Editar el estado actual de un dispositivo, ya sea de luz o climatización

La función que dispone el cliente móvil, además de las comunes, es realizar una programación sobre un dispositivo en específico, en fecha y hora, de tal forma que se realiza una cambio de estado del dispositivo en la fecha y hora programadas. Veamos una captura:

Como ya se ha comentado, el cliente web, por su parte, nos dispone de las funciones de administración, como el añadir y borrar dispositivos:



Estas operaciones no se encuentran en el cliente móvil por razones de comodidad y usabilidad. Conforme el proyecto escale se añadirán nuevos dispositivos y más configuraciones engorrosas y obtusas desde un dispositivo móvil, por lo que se prefiere dejar estos temas de configuración en un dispositivo más viable para tal propósito, como lo es un computador.

Con los cuatro elementos descritos ya tendríamos los componentes principales de la arquitectura resumidos y sabiendo cómo funcionan. Sin embargo esto ha llevado un proceso de desarrollo a lo largo de doce meses en el que el proyecto ha evolucionado, desde las primeras pruebas simulando dispositivos hasta encender una bombilla LED doméstica de 220W.

Pasemos a lo interesante de ésta memoria, el proceso completo de desarrollo.

PROCESO DE DESARROLLO

Para esta sección se narrará el proceso de manera cronológica, es decir, la evolución desde que el proyecto empezó hasta lo que es hoy en día. Únicamente se detallará el proceso de creación del software y hardware necesario para el funcionamiento, por lo que no se especificará nada en torno a ejemplos ya montados de la arquitectura como podría ser una maqueta simulando un entorno real.

Hay bastante que contar en este apartado, que estará repartido entre:

- La creación de la API REST en NodeJS
- La creación de un cliente Web en ReactJS
- La creación posterior de un cliente móvil de nuevo en ReactJS, de la mano de React-Native
- La puesta en marcha del controlador de la arquitectura Raspberry Pi 3
- Refinado de API REST para admitir parámetros de la Raspberry Pi
- La programación de la comunicación entre API y controlador Raspberry
- Refinado de los clientes móvil y web para la integración con la nueva API
- Programaciones de fecha y hora en dispositivos
- Refactorización e Integración Continua

Creación de API REST en NodeJS

La API, como bien hemos comentado antes, está escrita en NodeJS (ver más arriba), por lo que su desarrollo no ha sido el de una API tradicional como puede ser Lumen (framework PHP específico para API) o Go. Antes de entrar en materia, para ejecutar el servidor en Node podemos usar dos variantes en nuestra terminal:

- "npm start" ejecuta el servidor una sola vez, sin tener en cuenta futuros cambios
- "nodemon" reinicia el servidor cada vez que se efectúa un cambio en el código

Como es normal, se ha usado casi en todo el proyecto la segunda variante puesto que es la que más interesa a la hora del desarrollo. Más adelante veremos para que nos resulta útil la primera variante.

Lo primero que se hizo fue crear una base de datos de prueba. La base de datos era una de tipo relacional SQLite3 en un archivo local que, aunque se asemeja bastante a MySQL, no mantiene las mismas características. Por ejemplo, al efectuar un borrado, SQLite3 no tiene en cuenta dependencias de claves ajenas.

La base de datos en este punto constaba de estas entidades:

- Casas: Tienen uno o muchos controladores (Raspberrys) a su cargo. Pertenecen a un usuario
- Controladores: Pertenecen a una sola casa
- Dispositivos: Pertenecen a un controlador. En este punto únicamente se emulaban dispositivos de climatización, con una temperatura y un nombre
- Programaciones: Establecen una programación en formato DATETIME para un dispositivo con la acción
- Usuarios: Guardan las credenciales de inicio de sesión. Un usuario puede tener varias casas o ninguna

Además al principio la inicialización de la BBDD no se hacía mediante un script ajeno al servidor, si no que se hacía al iniciar el propio servidor, por lo que en cada reinicio se ejecutaba de nuevo el código que limpiaba la BBDD y la volvía a poner con valores iniciales. Esto es, en esencia, una mala práctica que no debería haberse hecho ni siquiera para pruebas rápidas. Se hizo por desconocimiento, ya que la experiencia en Node era nula y no se quería perder tiempo tampoco configurando bases de datos. Todo esto con la librería sqlite3 de npm.

Por otro lado, el proceso de manejo de datos se realizó con la librería knex de npm. Con ésta simplemente se necesita una mínima configuración del driver de la BBDD para poder funcionar. Luego de esta pequeña configuración se nos abstrae de cualquier proceso de datos, por ejemplo podemos hacer una sentencia SELECT sin importar que motor de base de datos estemos usando.

Con la librería knex, únicamente debemos emplear las funciones dadas por la API de la librería para poder hacer uso de ella sin importarnos el driver que estemos usando. Por supuesto, según el driver que estemos usando no será la misma configuración, puesto que hay motores de BBDD que requieren algunos parámetros especiales.

Siguiendo con las funciones de BBDD, en este punto del proyecto se apartaron todas las funciones de este tipo, dejándolas en funciones específicas que trataran con los datos y los devolvieran el funciones “callback”. Las funciones callback son

necesarias puesto que el funcionamiento de NodeJS, que al fin y al cabo es JavaScript, es asíncrono. Las líneas de código no se ejecutan siempre en el mismo orden, y con las funciones callback lo que hacemos es, en esencia, hacer el código un tanto más síncrono. Para que se comprenda: hasta que una función callback no sea ejecutada no se puede pasar a otra sección de código.

Esto va a ser útil durante todo el proyecto, tanto en la API como en los clientes JavaScript posteriormente comentados. Para cada ruta de la API se ha definido un endpoint que al final devuelve un código de estado HTTP, normalmente 200 OK, 404 NOT FOUND o también códigos especiales, como el 401 Not Authorized o el 201 CREATED si estamos dando de alta un nuevo registro en BBDD. Además de devolver códigos de estados también podemos devolver un cuerpo de mensaje, normalmente en formato JSON.

Para montar la función de API se han utilizado la librería de express, junto con el apoyo de las librerías de url y body-parser para poder leer bien cuerpos JSON y URL-encoded que puedan venir en las peticiones a la API. Con la librería de express, previamente configurada con un puerto puesto en código, simplemente con una exportación sería posible usar ya la API.

Con la librería de express se definió una variable que permitió enrutar todos los endpoints en funciones separadas para facilitar su manejo y lectura. Como ya se ha comentado antes, para cualquier llamada a una función ajena que espera datos, se necesita una función callback para asegurarnos que procesamos dichos datos cuando estén preparados, y no antes. Es por ello que se ha seguido durante todo el proyecto el mismo proceso para validar peticiones: función callback, se validan los datos de dicha función, se pasa a nueva función callback. Si no se encuentran dichos datos o son inválidos se manda un mensaje de error con el código pertinente.

Hay funciones que requieren de una autenticación especial, como son las PUT/POST/DELETE, que crean, modifican y borran datos en base de datos. En la ruta de login se devuelve un código único (token) que se crea con los datos de inicio de sesión únicos de cada usuario (nombre de usuario + fecha de expiración). Este código único se crea con el estándar JWT (JSON Web Token) y se devuelve en el cuerpo de la función si el inicio de sesión ha sido satisfactorio.

JWT, con los parámetros pasados, crea un código único que al devolverse en una función callback se devuelven al cliente en forma de cuerpo JSON. Dicho código único se ha de pasar a cada ruta comentada anteriormente, en caso de no ser así se envía un mensaje de error 401 indicando que la autorización no es pertinente, junto con la url de login pertinente. Esto también ocurre no solo si no pasamos dicho código, si no que también si el código que se ha pasado no es válido o ha expirado.

Todo esto se valida gracias a la función middleware puesta en cada ruta que precisa de este grado de autenticación. Esta función middleware valida el token pasado, extraído de la cabecera http "Authorization", que contiene el valor "Bearer " junto con el token del cliente, dando paso a la siguiente petición (la ruta que realmente deseamos procesar), o parando la ejecución de la petición pasando el mensaje de error al cliente.

En este apartado queda por comentar los endpoints que no necesitan de la cabecera de autenticación. Métodos GET cuya función es extraer datos de la BBDD para mostrárselos al usuario en forma de array JSON. Se ha pretendido hacer la API lo más RESTful posible, por lo que en los métodos GET, por ejemplo, se nos devuelven más campos además de los datos que queremos en sí: offset para la paginación de elementos dentro de un array para aumentar la eficiencia en la búsqueda y para paginado, urls de siguiente "página" dentro de un offset, url de edición/borrado de un elemento en concreto, etc.

Por ejemplo, para el endpoint GET de recibir los datos de un controlador en específico:
Url: http://localhost:8080/api/casas/1/controller/1?offset=1
Resultado:


```
{
    "id": "1",
    "nombre": "Vestíbulo",
    "casa_id": "1",
    "dispositivos": [
        {
            "dispositivo_id": 2,
```

```
            "nombre": "Aire acondicionado",
            "temperatura": 24,
            "url":
"http://localhost:8080/casa/1/controller/1/regulador/2"
        },
...
],
    "anyadir_dispositivo":
"http://localhost:8080/casa/1/controller/1"
}
```

Si no usamos parámetros como el offset no importa, obtendremos todos los registros en vez de paginar los resultados. En estos momentos la aplicación pagina de cinco en cinco. En futuras versiones se podría implementar el paginado según se le pase un parámetro adicional "limit" que indique el número de registros que queramos obtener.

Por último se implementaron una serie de tests, separados en un directorio distinto, para ejecutarlos después de cada commit con la librería mocha y supertest, de npm, que realiza peticiones a nuestros endpoints y evalúa la respuesta de cada una, tanto el código como la respuesta JSON que se obtenga.

Con todo esto teníamos una versión inicial de la API con la que hacer pruebas en los clientes.

Creación de Cliente Web en ReactJS

ReactJS, pese a que es una potente librería JavaScript para la creación de clientes web, puede ser un auténtico caos mantener el proyecto si no se mantiene el código ordenado.

Es por esto que lo primero que se hizo fue organizar el código en directorios bien estructurados, siguiendo las directrices de la mayoría de desarrolladores JavaScript como por ejemplo, Dan Abramov (@dan_abramov en Twitter). Dichas directrices se resumen en unos cuantos directorios:

- src: Es el directorio principal donde se guarda todo lo relacionado con desarrollo de nuestro código. Todos los directorios comentados a partir de aquí se guardarán en src.
- API: Aquí se guarda todo lo relacionado con los métodos y URLs de la API desarrollada (apartado anterior).
- components: Componentes React de nuestra aplicación. No olvidemos que el potencial de React es la reutilización de componentes independientes, como por ejemplo una imagen, un conjunto de labels, etc.
- views: Aquí se almacenan las vistas principales que se cargan según la ruta del navegador. Son al fin y al cabo componentes de la aplicación.
- images: Directorio con imágenes usadas en la web.
- store: Se almacenan los ficheros JavaScript necesarios para Redux (explicado más adelante).
- reducers: Se almacenan los ficheros JavaScript necesarios para almacenarlos en el estado con Redux (explicado más adelante).
- containers: Directorio con las vistas principales cargadas desde un archivo JavaScript para mantener el estado con Redux.
- actions: Redux precisa de unas acciones para cambiar el estado de la aplicación, que están guardadas en archivos JavaScript dentro de este directorio.

Y con este "par" de directorios se podría trabajar bien pudiendo mantener el código cohesionado y desacoplado para que el proyecto pueda tener una esperanza de vida al menos, no prematura.

Antes de ponernos en materia deberíamos explicar un poco la funcionalidad de Redux dentro de una aplicación escrita en React, ya que lo vamos a escuchar bastante a lo largo de este proyecto. Muchas veces tenemos la necesidad de mantener el estado, por ejemplo, de la sesión de un usuario para verificar que páginas puede visitar y cuales no. No tiene sentido que un usuario logueado pueda volver a iniciar sesión ¿verdad? Aunque podríamos pasar en cada vista y componente el estado de forma manual, Redux nos abstrae de todo esto, pasándolo directamente como propiedades desde los archivos container, haciendo que el estado sea global a toda la aplicación.

En el inicio de sesión le estamos pasando a la vista de login el estado que queremos a las "props", que son unas propiedades de React. Por ejemplo, si queremos acceder a la variable "user" desde cualquier componente (previamente conectado con Redux), únicamente debemos acceder al estado mediante "this.state.user".

Centrándonos en la función "login", la importamos desde el directorio actions, mencionado anteriormente. Las acciones por su parte son las funciones que cambian el estado en el que se encuentra, por ejemplo, la sesión del usuario. Dichas funciones se exportan para poderlas conectar a Redux, como ya hemos visto en el container. Dichas acciones se reducen (valga la redundancia) en archivos dentro del directorio reducers que definen la estructura de cada función de dicho reducer para Redux. Hay reducers para la autenticación, otros para la navegación, otros para alertas…

Pongamos un ejemplo de nuestro reducer y action respectivamente de la autenticación:

```
import { LOGIN, LOGOUT } from '../actions/auth';
const initialState = {
  error: false,
  user: []
}
export default function reducer(state = initialState.user,
action) {
  switch (action.type) {
    case LOGIN:
      return Object.assign({}, state, {
        user: action.user
      });
      break;
    case LOGOUT:
      return Object.assign({}, state, {
        user: action.user
      });
    default:
```

```
      return state
  }
}
```

Como vemos, al crearse el estado hay un estado inicial, en el que el usuario está deslogueado y por supuesto es vacío. Tenemos dos casos, uno para el login y otro para el logout, en el que asignamos al usuario el nuevo estado, normalmente el usuario para el login y el usuario vacío para el logout.

```
import { signIn } from '../API/methods';
export const LOGIN = 'LOGIN';
export const LOGOUT = 'LOGOUT';
const loginSuccess = (user) => {
  return {
    type: LOGIN,
    user
  }
}
const logoutSuccess = () => {
  return {
    type: LOGOUT
  }
}
export const login = (username, password) => {
  return function (dispatch)  {
    signIn({login: username, password:
password}).then((body) => {
      const user = {
        name: username,
        token: body.token
      }
      dispatch(loginSuccess(user));
    })
    // ...
```

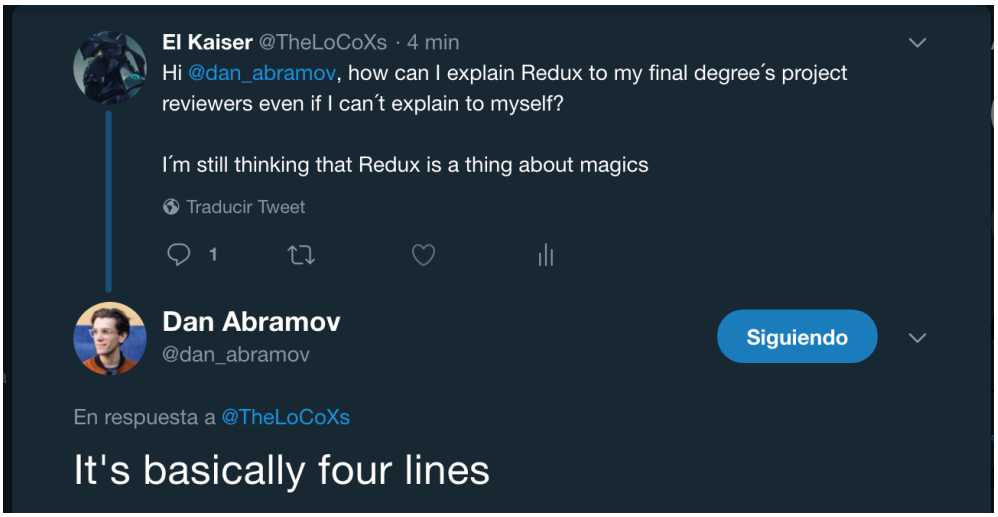
```
  };
}


export const logout = () => {
  return function (dispatch) {
dispatch(logoutSuccess());
  };
}
```

Como vemos, el código de actions para la autenticación tiene toda la lógica de negocio que se espera para dicha función. Si el login es satisfactorio se despacha el usuario, con su nombre de usuario y el token único (véase subapartado anterior), necesario para las operaciones delicadas con la API.

En general, para cada funcionalidad (autenticación, alertas, etc) se definen un reducer y un action, para luego ser fusionados en Redux en un único archivo que procesa cada reducer.

Pese a que el estado siempre ha sido un problema presente en el mundo JavaScript, Dan Abramov puso una solución muy convicente a una problemática que no hacía si no que producir código engorroso y de difícil mantenimiento. Esta genialidad, reconocida por muchos, "no son más que cuatro líneas", tal y como me indicó su creador al preguntarle cómo explicar Redux a los lectores de este proyecto:



Después de esta explicación (puede que algo confusa) de Redux, bastante complicado de explicar por cierto, pasemos a la interacción con la API. Pensando en el futuro se ha almacenado todo lo relacionado en un mismo directorio. En él podemos encontrar dos archivos, uno de urls y otro de métodos. El primero, como es obvio, contiene únicamente las dos urls base para la interacción con nuestra API, que usaremos en el código de los métodos.

El archivo de los métodos, por su parte, son funciones exportadas que representan un endpoint de la API cada una. Usamos una función "fetch" de JavaScript para realizar una petición AJAX a nuestra API. En nuestro "body" pasado por parámetros extraemos todos los datos necesarios.

Estas funciones son usadas, por ejemplo, en las vistas o componentes, para llamar a la API y poder recoger datos sin necesidad de implementación en el propio código de la vista. Además así estos métodos de API pueden ser reutilizados, como veremos más adelante.

Por otra parte, se ha usado la librería React-Router para la navegación, puesto que nos simplifica enormemente el tratado con urls y parámetros. Con un simple componente en el que definimos las rutas, todas exactas, podemos importarlo en el archivo principal App.js. Al final los componentes de React utilizan una función "render" que renderiza en tiempo real, según el estado del componente, el código escrito en dicha función. Es por esto que únicamente renderizamos el componente "Router" implementado, desde el que se renderizarán los componentes pertinentes.

Visualmente se ha seguido la estructura de "una ruta específica pertenece a una vista específica", es decir, una vista engloba varios componentes, además de poder tener elementos propios (títulos, navbar, etc). A estas alturas del proyecto tenemos cuatro vistas: login, casas, controlador y dispositivos de un controlador.

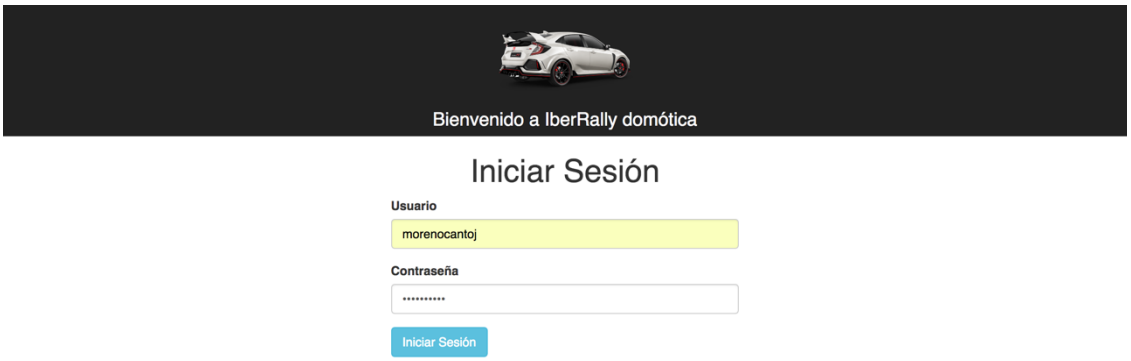
Para esta versión el estilo visual era muy básico, bastante más enfocado a una parte administrador de la arquitectura. Lo importante era lo que no podíamos hacer

cómodamente con un móvil: añadir dispositivos, borrar dispositivos, cambiar parámetros de un dispositivo, etc. Así que el estilo visual aquí, poco importaba.

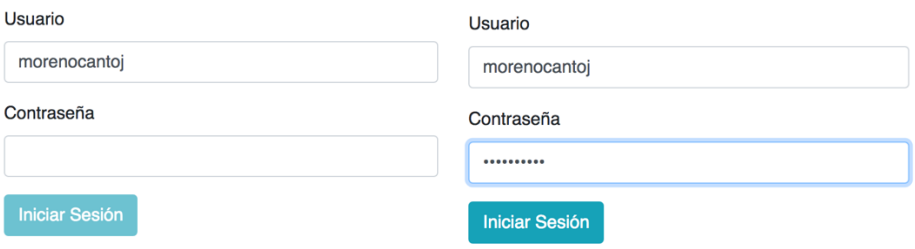
No obstante sí que se han creado archivos de estilo CSS para componentes y uno principal para las vistas, para básicamente unificar todo el estilo común. Además en este punto del cliente también se tenían validadores de campos, por ejemplo en el componente de login.

Temas como los validadores en React pueden ser muy tediosos si no se entiende bien, pero en la teoría son muy simples. En el estado hay una propiedad que indica si el formulario es válido. Si dicha propiedad está en falso el formulario no puede ser completado, mientras que si el formulario tiene todos los datos correctamente dicha propiedad pasa a ser verdadera, pudiendo completar el formulario satisfactoriamente.

Por ejemplo, los validadores para el formulario de inicio de sesión son que tanto el usuario como la contraseña sean rellenados, sean correctos o no eso lo decidirá el método de API correspondiente:



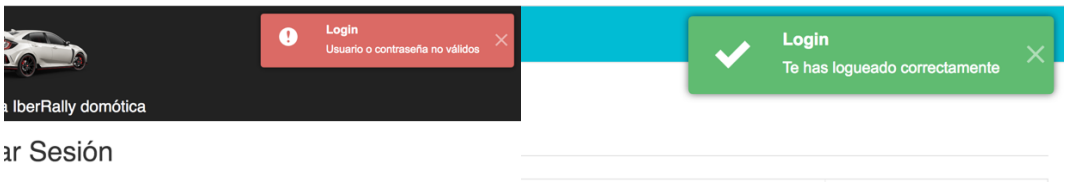
Al intentar hacer login sin poner ambos campos, podemos observar que el estado de la propiedad válida del botón de iniciar sesión sigue estando en falso, ya que no hemos acabado de introducir las credenciales. No ocurre lo mismo cuando tenemos sendos campos introducidos:



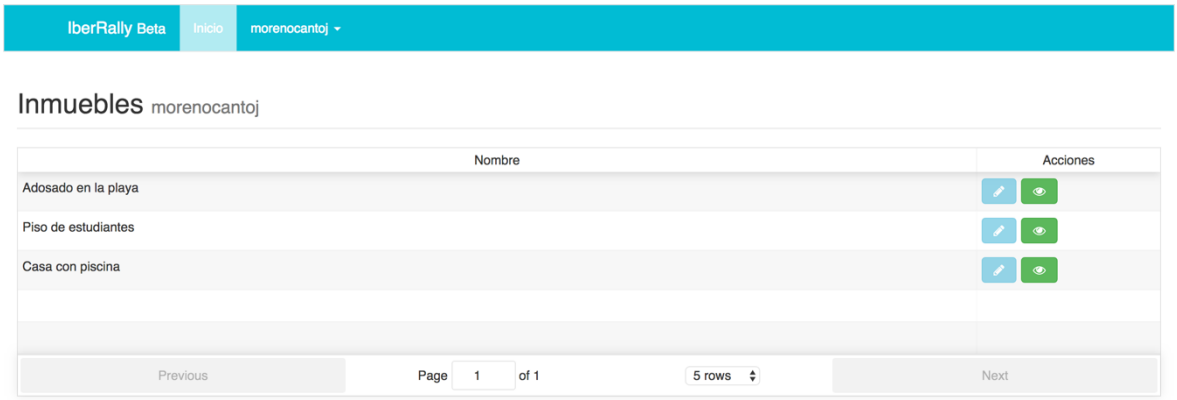
Lo que está ocurriendo aquí es que el lenguaje JavaScript está haciendo su propia magia. Cada vez que el usuario cambia uno de los campos ambos se evalúan, ya que cada campo tiene su propia propiedad de estado indicando si el campo en cuestión es válido. En este caso las propiedades de cada campo son válidas si el tamaño del input es mayor que cero, es decir, si hay algo escrito. Por ende, si ambos campos son válidos, el formulario en sí es válido, desbloqueándose así el formulario.

Para validaciones de respuestas de la API se ha optado por otro sistema. Ya que debemos esperar a la respuesta de la API, lo que nos devuelva la misma será lo que evaluemos a la hora de verificar si la petición se ha procesado correctamente o no. Para mostrarlo de cara al usuario, se ha adjuntado al estado de la aplicación (Redux), además de la autenticación, la librería toastr, que nos permite mostrar avisos en tiempo real sin necesidad de recargar la página. Esta pequeña función de alerta es muchísimo más sencillo de implementar que cuando usamos lenguajes server-side, como PHP.

Por ejemplo, si iniciamos mal la sesión, nos aparecerá una alerta de error, mientras que si la petición en la API ha ido bien, se nos guardará el estado del usuario en Redux y nos aparecerá una alerta de que todo ha ido correctamente:



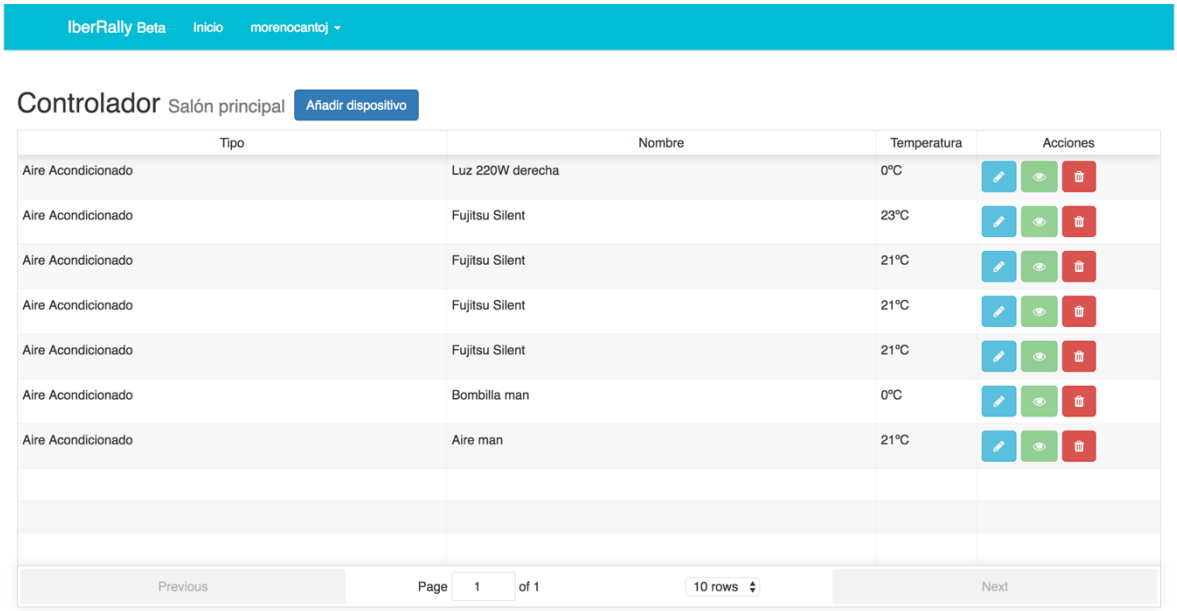
Una vez tengamos la sesión iniciada es imposible volver a la pantalla de inicio de sesión si no es apretando al botón de "cerrar sesión" de la barra de navegación superior que nos dispone la aplicación. Esto es gracias a que Redux mantiene el estado de nuestro usuario logueado y nos impide volver hacia atrás. Es más, si volvemos a la raíz de nuestra web se nos redirige directamente hacia la pantalla principal del usuario logueado, que es un listado de sus casas:



Como hemos comentado antes, la interfaz es muy básica. Disponemos de una barra de navegación superior, con el link de inicio (preseleccionado puesto que estamos en dicha sección), el nombre de la aplicación que redirige a la raíz (en nuestro caso la sección Home), y nuestro nombre de usuario, indicándonos acciones pertinentes a la entidad, como por ejemplo cambiar ajustes o cerrar sesión.

Muchos de los botones y links están deshabilitados, puesto que para esta versión no estaban pensados de implementar, dejando constancia de que sí se harían en un futuro. Con una propiedad CSS de bootstrap lo hacemos posible sin mayor complicación. El listado de casas se hace mediante un componente descargado de npm, que no es más que una tabla dinámica configurable por el usuario tanto en filas por página, campos a mostrar, render personalizado por columna, estilado, etc. Lo bueno de React es que hay infinidad de componentes ya implementados y configurables, lo que nos ahorra mucho trabajo.

La vista de controladores no la vamos a comentar puesto que sigue la misma filosofía que la página de inmuebles. No obstante si entramos en el detalle de un controlador (recordemos que un controlador representa una Raspberry Pi en nuestro inmueble), tenemos nuevas funcionalidades.

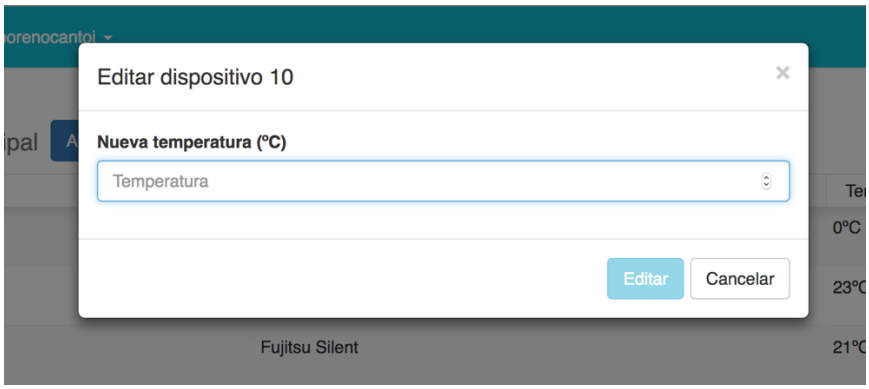


Como vemos, la tabla tiene más campos además de más acciones. Para los iconos se ha usado la librería Font Awesome. Dentro de esta página podemos añadir un dispositivo, borrar uno ya existente en la tabla o simplemente editar los parámetros de los mismos. Para cada acción se abrirá un modal de Bootstrap que comentaremos posteriormente.

Los dispositivos que aparecen son extraídos bajo una llamada a nuestra API. Como podemos prever, los parámetros de un dispositivo pueden cambiar mientras estemos nosotros en la pantalla. Es más, podría incluso borrarse un dispositivo. Para ello se tuvo en cuenta el estado del componente de los dispositivos. React renderiza en tiempo real, pero solo vuelve a renderizar el componente cuyo estado ha sido modificado. La llamada a la API devuelve un listado de dispositivos, que son almacenados en el estado. Si tan solo uno de esos dispositivos cambian, el estado del componente de dispositivos cambia, por tanto, el componente se volvería a renderizar.

Para verificar si el estado de los dispositivos ha cambiado necesitamos una segunda llamada al método GET de la API. Para ello usamos la estrategia de polling. Usando un componente de npm podemos configurar un refresco a la llamada a la API en el tiempo que estimemos oportuno. En este caso el timeout establecido para el polling ha sido de diez segundos, tiempo más que suficiente para que la API no se sature ante posibles miles de clientes haciendo polling a la vez.

Pasemos a los modales anteriormente mencionados. Los modales en las librerías JavaScript suponen una gran ventaja frente a los lenguajes server-side. Mientras que en estos últimos lo cómodo para el desarrollador es realizar un modal por cada fila de una tabla, lo que supone más tiempo de carga de la página. React, por su parte, permite definir un solo modal para cada acción, pudiendo activarse en cada fila simplemente asignando un cambio de estado, por ejemplo, del dispositivo seleccionado, siendo dicho modal quien recoja ese cambio de estado y recogiendo ese dispositivo seleccionado. Lo podemos hacer con simples posiciones de array como lo podemos hacer metiendo un elemento seleccionado entero en el estado, con todas sus propiedades. Pongamos un ejemplo, editando un dispositivo:



Como vemos, tenemos de nuevo validadores, en este caso dos. Primero se valida que el formulario sea válido, para ello debe haber algún número puesto en el input de temperatura. Además, al ser un input de tipo número, no deja poner valores incorrectos para una temperatura, como podrían ser letras. En caso de no querer editar el dispositivo siempre podemos cerrar el modal clickando en "Cancelar" o en la propia cruz en la parte superior derecha.

Con los modales se tuvo un ligero problema: el polling a la API provocaba un bug que quitaba el modal sin quitar la transparencia. Además, si justo se cambiaba el dispositivo que estábamos editando o borrando podría haber un conflicto en la transacción. La solución al problema fue simplemente poner una propiedad en el estado indicando si estábamos en algún modal. Cuando dicha propiedad se ponía a verdadero el componente de intervalo que realizaba el polling se paraba, por lo que no se refrescaba la lista de dispositivos.

No obstante, como es normal, al editar/borrar/crear cualquier dispositivo se volvía a llamar al método pertinente de los dispositivos para refrescar el estado de los dispositivos. Al ser métodos completamente separados, modificar los dispositivos no cambiaba el estado del listado de los mismos, por lo que no se actualizaba la tabla.

Para finalizar esta subsección, un tanto larga por la explicación de Redux y todas las imágenes mostradas, recordamos que esta versión del cliente web es prematura, y como bien indica la barra de navegación, es una beta, desarrollada durante el período de la asignatura de ADI pensada ya para el proyecto presente, por lo que se incluye aquí también.

Creación de Cliente Móvil en React-Native

Como ya se ha comentado, React-Native es una librería muy semejante a React, pensada para construir aplicaciones móviles multiplataforma. Esta sección no será tan larga puesto que casi todo el trabajo se ha hecho en la sección anterior, creando el cliente web.

Empezemos por lo común al cliente web en React. Básicamente, toda la implementación de Redux y la integración con los métodos de la API se ha copiado y pegado desde el proyecto web al móvil, puesto que el código funciona de la misma manera en ambos al ser React en esencia. Además el comportamiento lógico de la aplicación es el mismo, solo que esta vez enfocado a una interfaz móvil. Además la estructura de directorios es exactamente la misma, a excepción de que las carpetas se guardan directamente en raíz, al contrario de cómo lo hacíamos con el cliente web.

Teniendo la mitad del trabajo adelantado gracias a compartir librería de desarrollo JavaScript, hay algunos apuntes que se deben de matizar respecto a las diferencias entre React y React-Native. La navegación móvil se basa en un sistema de pila, donde hay pantallas principales en las que se van apilando pantallas encima, para cuando volvemos a atrás ir sacando elementos de la pila (pop/push). React-Navigation es la librería de npm encargada de sustituir a la navegación entre páginas de React-Router por la navegación por pila de React-Native.

Costó bastante este apartado puesto que se integró con Redux, ya que, como hemos comentado anteriormente, un usuario logueado no debería poder volver a la pantalla del login si no se ha desconectado de la sesión antes. Manteniendo el estado hemos podido evitar que esto ocurriera. Los recientes cambios en la API de React-Navigation hicieron que la búsqueda de información resultara obsoleta, por lo que dificultó un tanto el avance del proyecto.

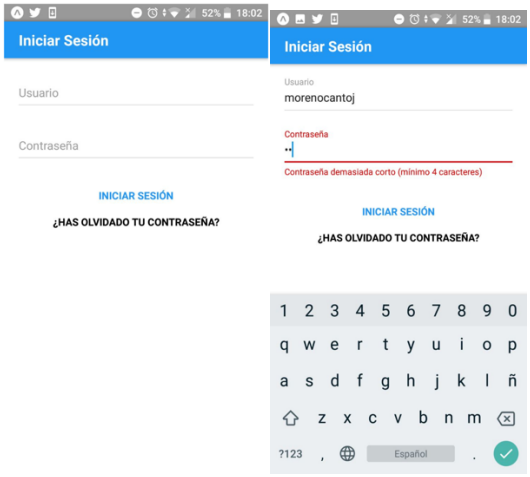
React-Navigation por su parte tiene dos tipos de navegaciones: por pantalla (stackNavigator) y por tabs (tabNavigator). La filosofía para la navegación es, en un componente aislado (igual que con React-Router), programar la lógica de pantallas junto con sus parámetros (nombre, barra de navegación, etc). Para ello se crea una estructura JSON que define dicha lógica en un stackNavigator, cargando en cada página el container correspondiente (recordemos lo anterior explicado de Redux). En vistas aisladas también se ha usado una estructura nueva de tipo tabNavigator, como veremos en un futuro. Sin embargo, esta última no se ha integrado con Redux.

Sin embargo, con React-Navigation se tiene que pasar en cada container las diferentes acciones que hagan los botones físicos (o virtuales) del Smartphone. Por ejemplo, si se quiere volver atrás, tanto con un botón en la vista como con el botón físico, dicho comportamiento se tiene se programar, según el sistema operativo, en una función "listener" que se activa cuando pulsamos dicho botón, ya sea hardware o en la propia vista. La función que va adscrita no es más que un método del stackNavigator que "popea" la vista actual, dando paso a la anterior que estábamos.

Pasando a temas visuales, se ha intentado adecuar la interfaz lo máximo posible para una persona adulta, tanto para jóvenes como adultos ya de mediana edad. Para ello haremos uso del estándar, sobretodo móvil, Material UI, que es bastante utilizado en la actualidad en cuanto a directrices para el desarrollo de una interfaz móvil.

React-Native mantiene unas diferencias en cuanto a ReactJS para el estilado de sus vistas. Mientras que en ReactJS tenemos archivos CSS, en React-Native debemos crear una hoja de estilos como una estructura JSON, ya que se comportan como propiedades dentro de los componentes de React-Native. Además no podemos poner texto aislado, si no que debe estar encapsulado dentro de los componentes de React-Native dentro del método "render".

Se ha elegido un tema de azules y blancos para el desarrollo de la aplicación móvil con Material UI. El tema de los validadores se comporta de igual forma que en ReactJS, por lo que también se ha duplicado. Por ejemplo en el formulario de inicio de sesión:

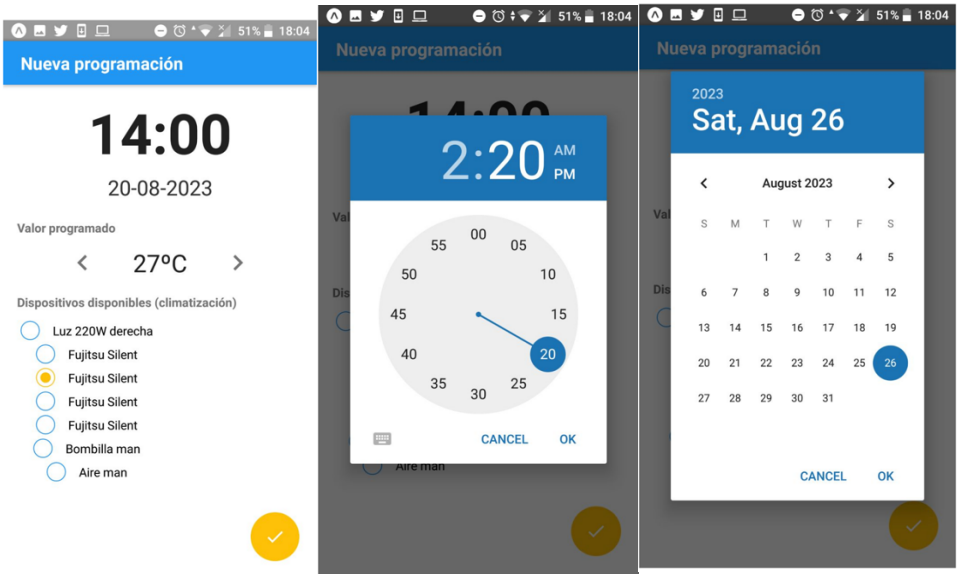


Simplemente se han cambiado las alertas de la librería toastr por alertas nativas, según estemos en Android o en iOS, que deberemos cerrar manualmente.
Como cambios significativos respecto a la aplicación web, es que esta app móvil se ha pensado para el uso más cotidiano del sistema, es decir, para la funcionalidad de la domótica en sí. Es por eso que con la app móvil podemos hacer estas acciones (en esta versión):

- Iniciar sesión y ver los inmuebles que tenemos disponibles (Web y móvil)
- Ver los controladores (Raspberry Pi) asociados a un inmueble (Web y móvil)
- Ver los dispositivos, junto con sus valores, conectados a un controlador (Web y móvil)
- Cambiar el valor de un dispositivo en tiempo real (Web y móvil)
- **Crear una programación para una fecha y hora en concreto en un dispositivo (Únicamente móvil)**

La realización de las programaciones es una llamada más a nuestra API, que por el momento únicamente apunta las programaciones en base de datos, por lo que la lógica de cómo se hagan no nos importa. Simplemente pasamos un dispositivo, una fecha y una hora, de modo que se pueda (en un futuro) realizar dicha acción.

Gracias a usar React se han podido reutilizar componentes propios que recogen los datos de un dispositivo, actualmente emulando climatización solo, para así usarlos tanto en la configuración del dispositivo en el momento como en la programación.



Pese a que sea una versión temprana de la aplicación móvil, se ha querido dejar preparada para en el futuro implementar más funcionalidades (diferentes dispositivos, estilos visuales dependiendo de los mismos, etc). Como vemos ya es una versión más profesionalizada que la versión web que teníamos en la versión anterior.

Puesta en marcha de un Controlador Raspberry Pi 3

Por supuesto antes de nada hemos de instalar un sistema operativo ARM en nuestra Raspberry Pi 3. En nuestro caso ha sido un Debian adaptado desde la propia web oficial de Raspberry Pi Project.

Antes de empezar a programar una aplicación para el manejo de dispositivos desde la Raspberry Pi, debemos tener claro qué es lo que hace la Raspberry Pi y cómo lo hace. Antes que nada se han instalado todas las dependencias y actualizaciones que requiere nuestro sistema operativo Raspian. Al no aceptar Sublime Text, ya que no es válido para procesadores ARM, se ha instalado Visual Studio Code.

Para iniciar esta sección hemos de saber primero cómo funciona la lógica para activar un pequeño LED en la Raspberry Pi, explicada anteriormente en el apartado de "Arquitectura del Sistema".

Como se ha explicado anteriormente (ver Herramientas y Tecnologías), la Raspberry Pi consta de varios puertos:

- 3,3V: Es el puerto lógico del procesador de la Raspberry Pi (1s o 0s) según el polo
- 5V: Es el puerto de voltaje necesario para activar, por ejemplo, un relé lógico que active una bombilla de 220W (+/- 5V según el polo)
- GPIO: Puerto de conexión de dispositivos para activarlos (1 o 0). Se activan mediante software desde la propia Raspberry Pi. Son, los que en esencia activan nuestros dispositivos, en este caso el pequeño LED de prueba
- GND: Puertos de toma a tierra

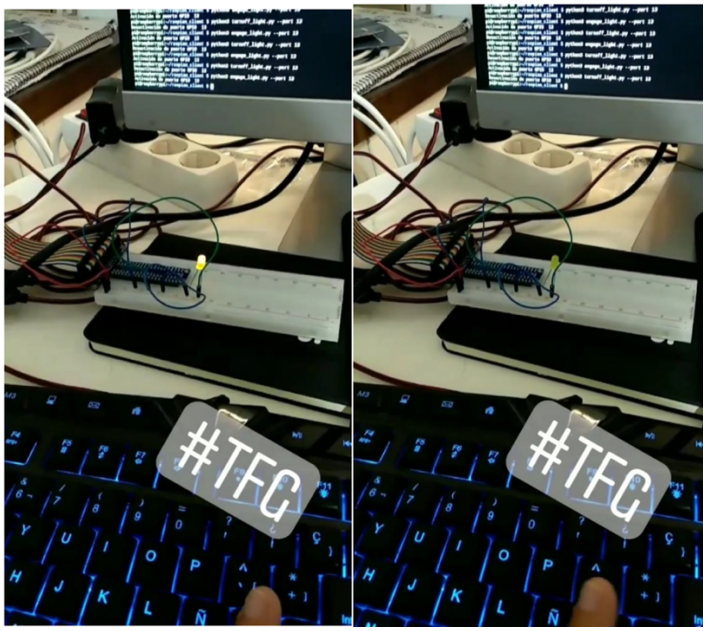
Para hacer funcionar un pequeño LED, únicamente debemos activar un puerto GPIO. Es lo que vamos a intentar hacer por ahora a estas alturas del proyecto. Para ello, el LED cuenta con dos varillas: la más larga se conecta a una clavija de la tabla de conexiones donde conectaremos un cable macho/macho, y la otra clavija la separaremos en otra altura (sin establecer contacto con los puertos expandidos de la Raspberry Pi).

En la misma fila que la clavija larga, como ya hemos comentado, se conectará un cable macho/macho que conectará un puerto GPIO, por ejemplo el 21, de la Raspberry con el LED (cable azul). La otra clavija irá directamente a una toma a tierra de la Raspberry Pi (cable blanco), ya que no disponemos de resistencias bajas para este experimento.

Una vez todo a punto, debemos elaborar el script que active y desactive dicho puerto. Para ello se ha elegido Python, ya que queremos un mismo script para activar y desactivar todos los puertos pertinentes, además de que cuenta con librerías para ello y para comunicarse con nuestra API, como veremos más adelante.

Lo primero que se hizo con Python fue un pequeño script donde se le pasaban por parámetros en terminal el puerto que se deseaba activar, ídem para desactivar, que estaba separado a su vez en otro script diferente, pese a ser el código casi idéntico al script de activar.

Los resultados fueron, después de múltiples intentos, gracias a la altísima resistencia que se puso, insatisfactorios. Para ello se cambió de la resistencia a una toma directa a tierra. El LED por fin se encendía y apagaba según se ejecutaban los dos scripts, tal y como podemos ver en estas dos imágenes de mi propia galería de Instagram:



Con todo esto, nos quedaría adaptar la base de datos y la API para que los dispositivos se guardaran de otra forma, con los siguientes parámetros:

- Tipo: puede ser un dispositivo de climatización o una luz, aunque únicamente se maneje luz con la Raspberry
- Puerto: representación del puerto GPIO de nuestro controlador
- Estado: activado/desactivado

En los siguientes apartados explicaremos todas las modificaciones en API que se hicieron para adaptar estos nuevos campos al sistema, así como también la comunicación entre el controlador y la Raspberry Pi.

Refinado de API para admitir parámetros de Raspberry Pi

Como acabamos de ver ahora, los parámetros que en este punto teníamos en los dispositivos, actualmente de climatización, resultaban insuficientes para poder efectuar cambios a nivel hardware a la hora de, por ejemplo, encender una bombilla (en estos momentos un LED). Además ya se estaba pensando en una subida a producción para una primera versión, por lo que se cambió toda la estructura de base de datos a un servicio MySQL local, eliminando así SQLite de nuestro sistema, así como las funciones que reinicializaban la BBDD en cada ejecución del servidor.

Es por esto que se hicieron cambios en la tabla de dispositivos de la BBDD. Se optó por mantener una misma tabla dispositivos con los dos tipos que habían: climatización y el nuevo de luz, diferenciados por un campo expreso para indicar el tipo de dispositivo que era cada nuevo registro en dicha tabla. Se optó por este método para dejar la llamada a todos los dispositivos en un solo endpoint, ya que de otra manera se hubieran tenido que realizar al menos dos llamadas distintas. Si queremos los dispositivos de un tipo en concreto deberemos comprobar su tipo en el array devuelto. En un futuro se planea implementar un filtrado por parámetros en la url de la llamada al endpoint, abstrayendo al cliente de tal labor.

De esta forma también se modificó el endpoint PUT que cambiaba el estado de un dispositivo, para que, según qué tipo de dispositivo, cambiara o bien la temperatura, o bien el estado del puerto (true/false).

Ahora sí, en este punto nos queda pensar en la comunicación bidireccional entre el servidor NodeJS y cada controlador, punto resuelto en la siguiente sección.

Comunicación entre API y controlador Raspberry Pi

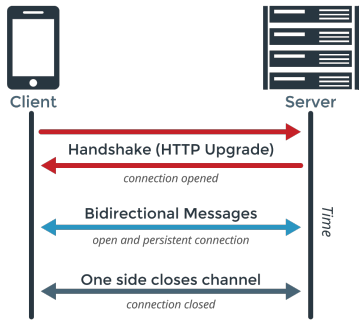
Como ya se ha comentado en el correspondiente apartado de Herramientas, la comunicación entre el servidor API REST y cualquier controlador se realiza mediante el protocolo websockets.

En un principio se quería usar la técnica que se ha usado en los clientes JavaScript, el "long-polling". Sin embargo, esta opción no es para nada beneficiosa para la programación de un controlador que debe estar siempre atento a cualquier cambio sobre un dispositivo que maneja.

Con la técnica de long-polling tenemos dos grandes problemas. El quizás un tanto menor es el tiempo de espera entre refresco y refresco. Pongamos que tenemos tres segundos de refresco a un endpoint de la API en específico para ver si algún dispositivo que manejamos ha cambiado de estado. Esto primero implica que debemos tener en la API un registro, ya sea local o en base de datos, de los dispositivos que han cambiado de un controlador en específico, borrando dichos registros según los controladores hayan verificado sus dispositivos. A esto se le suma la problemática de que, por ejemplo, si queremos activar cualquier luz, no es hasta el refresco de la API cuando nos damos cuenta. Esto quiere decir que si nuestro refresco es de tres segundos, y se ha activado la luz en el primer segundo, no se activará hasta dos segundos después. Poco tiempo en la teoría pero bastante mayor en la práctica, más si es reiterado.

El segundo y mayor problema es que tenemos una arquitectura de controladores descentralizada, por lo que cada controlador es independiente. Esto quiere decir que cada controlador de la casa, pongamos uno por habitación, está haciendo long-polling a la API. Cuantas más casas dispongan de este sistema, más controladores, lo que implica más polling a la API, probablemente la mayor parte innecesario. Esto a la larga provoca pérdidas de rendimiento abusivas en el servidor, que debe procesar cada petición. ¿Y si en vez de estar preguntando continuamente al servidor si algún dispositivo ha cambiado, hacemos que el servidor nos envíe un aviso cuando un dispositivo en concreto ha cambiado?

Esto es posible gracias al protocolo websockets. Tanto cliente (controlador) como servidor (API) son ahora cliente y servidor a la vez, ya que ambos envían y esperan recibir mensajes.



Para Python existe una librería de websockets básica, con la que se ha montado en un archivo aparte el cliente/servidor websockets. Para NodeJS se ha utilizado la librería Socket, montando el mismo entorno que con Python, solo que se ha fusionado con la API existente con Express, para que vaya por el mismo puerto y url.

La filosofía de websockets para nuestro caso es tal que así:

- Controlador envía un mensaje a la dirección del servidor indicando el ID de controlador que se está conectado
- Servidor responde al controlador y se guarda dicha conexión en una estructura de tipo mapa, donde el índice es el ID del controlador
- Cuando un cliente modifica un dispositivo, se coge el ID del controlador de la ruta del endpoint, pudiendo extraer el controlador pertinente del mapa almacenado localmente por el servidor, enviando el mensaje pertinente de cambio al controlador
- Controlador recibe el mensaje y efectúa cambios en local: activar/desactivar cierto puerto en nuestro caso
- Al desconectarse el controlador se envía un mensaje de desconexión al servidor, que borra dicho controlador de la estructura mapa comentada

Con esto, tenemos comunicación entre servidor y controlador por demanda, con lo que nos ahorramos infinidad de peticiones innecesarias, además de que el cambio efectuado es casi al instante. No obstante, se nos abre una nueva problemática: ¿Cómo sabe el controlador su propio ID?

De momento dicha problemática se solventa manualmente, ya que los controladores saben la dirección del servidor, pero el servidor en un principio desconoce la localización de los controladores, ya que para ello precisaríamos una IP estática de internet en la configuración de red en el inmueble, y esto sería más costoso y problemático a la larga. Cada controlador es dado de alta con un ID, independientemente del inmueble al que pertenezca. Dicho ID por el momento se ha de integrar a mano en el código, ya que el primer saludo del controlador a nuestra API se realiza con dicho ID, para así posteriormente en los endpoints saber que conexión websocket se ha de tratar.

Obviamente el sistema presentado precisa de una mano administradora a la hora de la instalación en el inmueble. De momento la solución es que el administrador sea el que configure el ID en el controlador la primera vez. En futuras versiones se intentará mejorar esta cuestión.

Una vez teniendo la comunicación y el encendido de LEDs a partir de la API (probando con los endpoints a mano por supuesto), se subió la API a la plataforma de Heroku, actualizando las url del código del controlador para así apuntar a la url de producción.

Antes de probar nada, se configuró un servicio externo de Heroku para tener una base de datos MySQL en producción, configurando el driver de knex para apuntar a dicha base de datos, inicializada gracias a nuestras copias locales de la BBDD de MySQL.

Una vez con todo configurado obtuvimos un problema, y es que Heroku corta las conexiones websocket a los 50 segundos de no obtener respuesta por uno de los extremos. Este pequeño problema se solventaba haciendo un pequeño ping desde el controlador cada 30 segundos al servidor, para que así Heroku no cortara la conexión sin que realmente acabara. Esta práctica se obtuvo de la propia documentación de Heroku en lo referente a la comunicación por websockets.

Una vez con nuestra arquitectura principal puesta en producción, llega el turno de que los clientes web y móvil se adapten a los nuevos cambios de la misma.

Refinado de los clientes móvil y web para la integración con la nueva API

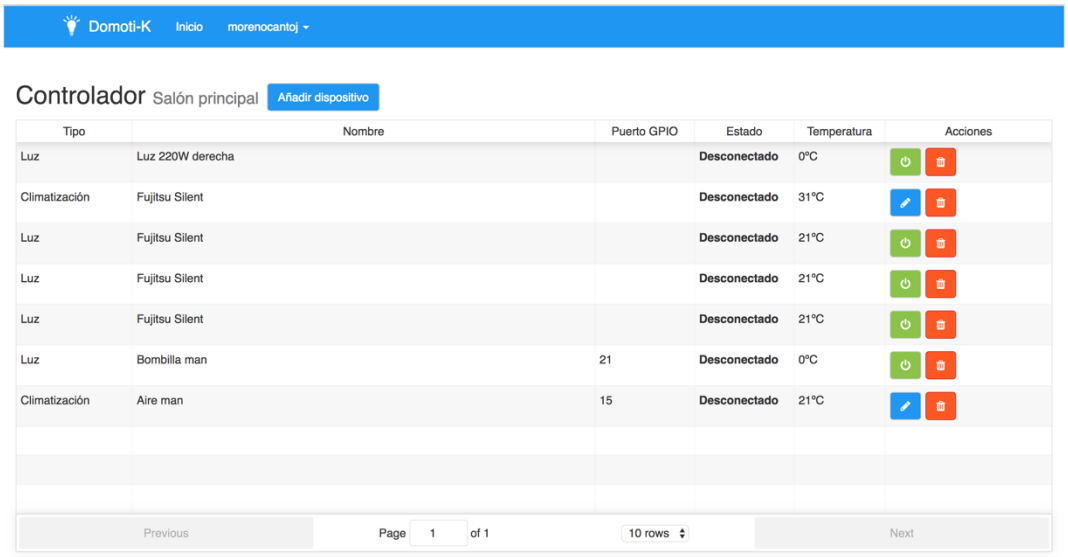
Al efectuar cambios como el diferenciado de tipo de dispositivos en la API, ambos clientes necesitaban ligeros cambios a la hora de tratar a un dispositivo. Lo que se realizó común a las dos plataformas fue cambiar los métodos de API a la hora de crear un dispositivo, para adaptar en el cuerpo JSON el tipo de dispositivo y el puerto GPIO correspondiente.

En el cliente móvil únicamente se cambiaron los iconos en el listado de dispositivos y se implementó un botón en la propia lista para poder activar/desactivar dispositivos de tipo luz, sin necesidad de más pantallas innecesarias. Además se

añadió un campo mostrando el valor, junto con colores representativos, del estado del dispositivo:

No obstante en el cliente web se hicieron bastantes más cambios visuales para asemejarse mejor a un producto profesional, además de los ya mencionados en la parte móvil para, aquí también, adecuar a según el dispositivo.

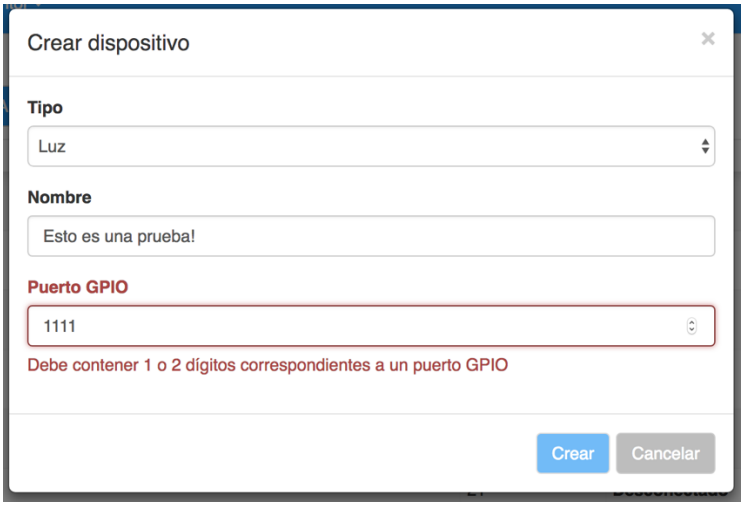
Se rediseñó la temática visual para ajustarse, al igual que la aplicación móvil, a las directrices de diseño de Material UI. No obstante, al ser una web orientada a la administración, se han mantenido elementos complejos, como las tablas dinámicas. Veamos algunas capturas del resultado:



Ha sido una completa pena sacrificar al Honda Civic Type-R de la pantalla de login, pero se precisaba un cambio estético.

Si nos centramos en la última imagen, vemos que ya tenemos una clara diferenciación en los dispositivos. Al igual que en la aplicación móvil, ahorramos pasos innecesarios al usuario al poner un botón directo para activar/desactivar dispositivos de tipo luz, mientras que mantenemos el modal del cambio de temperatura para dispositivos de climatización (emulados).

Además al añadir un nuevo dispositivo ya podemos diferenciar entre ambos tipos, debiendo poner obligatoriamente el puerto GPIO correspondiente a la placa de nuestra Raspberry Pi y un nombre. Ambos campos están validados como podemos ver en la siguiente imagen:



Como vemos la aplicación web está más orientada a la administración, mientras que en la aplicación móvil está más pensada de cara al usuario final, ya que hay funciones que no comparten, como ya se ha comentado en apartados anteriores. Todo el tema de creación/modificación de la información de dispositivos recae en la parte web, mientras que otros asuntos relacionados con la interacción del inmueble con el usuario final recaen en la parte móvil, como se verá más adelante.

Programaciones de fecha y hora en dispositivos

Para este proyecto se propuso, además de encender bombillas con un móvil, encenderlas en una fecha y hora concreta. Al fin y al cabo lo único que debemos de hacer es activar el puerto pertinente en la fecha y hora que acordemos.

Para esto la programación se centra sobretodo en nuestro controlador. Lo primero que se pensó fue en usar librerías que operaran con la utilidad Crontab de los sistemas operativos Linux (recordemos que la Raspberry Pi va sobre un Debian adaptado). El problema es que la herramienta Crontab planea trabajos para una un tiempo en concreto, durante una frecuencia infinita, ya sea cada día, cada semana, cada mes o cada año, pero no para una fecha en concreto.

Para el problema se ha optado por abrir un nuevo hilo de ejecución, que con la librería "threading" de Python se puede prorrogar hasta una cantidad de segundos establecida en la fecha y hora recibida por websockets. Por ejemplo, si el controlador recibe que debe activar el puerto GPIO 23 en una hora, dicha hora se pasa a segundos, dando un resultado de 3600 segundos, que se pasaría a la variable correspondiente que iniciaría el temporizador, junto con una variable que englobaría la acción pertinente.

Esto tuvo en primer lugar un gran problema, y es que cuando realizábamos una programación de un dispositivo se paraba la ejecución de todo el controlador. Esto era porque estábamos parando el propio hilo principal de la ejecución, ya que al principio no creábamos otro hilo aparte para nuestra funcionalidad de la programación. Esto ocasionaba que las peticiones de encendido/apagado sobre dispositivos no funcionaran, ya que el controlador no era capaz de responder a la API mediante websockets.

No obstante esto nos trae el percance de que puede que se nos abran demasiados hilos simultáneos centrados en programaciones de dispositivos. Al ser un sistema descentralizado, donde el peso de una zona en concreto, y no un inmueble entero, recae sobre un controlador únicamente, no se cree que se pueda convertir en un problema en cuanto al soporte de hilos simultáneos de una Raspberry Pi 3.

Cambiando de tema, la API tuvo que ser retocada para poder diferenciar entre dispositivos a la hora de las programaciones. Para ello, según el dispositivo, se envía una cadena con una serie de instrucciones con un formato decodificado por el controlador al recibir el mensaje por websockets. Por ejemplo, para un dispositivo de luz sería:

```
PUT light [puerto] [estado]
```

El primer parámetro siempre es el mismo, es el método http pertinente, que en el caso de las programaciones es siempre PUT. El segundo pertenece al tipo de dispositivo que estamos manejando, en este caso una luz. Los siguientes parámetros corresponden a la luz, ya que son el puerto GPIO asociado y el estado, true (encendido) y false (apagado).

Esto a su vez se integró en la aplicación móvil, que sí está pensada para poder realizar programaciones en fecha y hora. No obstante el método de API de la petición para realizar una programación es un tanto diferente al enviado por la API al controlador. Por ejemplo:

```
{
"dispositivo_id": 13,
"fecha": "02-08-2018 19:50",
"action": "PUT light OFF"
}
```

El dispositivo se manda por el cuerpo de la petición, ya que es la base de datos la que contiene el puerto GPIO correspondiente a dicho dispositivo. El ID del controlador ya lo tenemos en la propia ruta. La fecha está en formato DD-MM-YYYY HH:mm, por lo que al ingresar en la API esta es transformada para el correcto envío al controlador. La acción como vemos también es diferente, puesto que simplemente pone el tipo de dispositivo (información redundante) y el nuevo estado que queremos conseguir. Todo esto está debidamente decodificado en un método de la API para ser enviado correctamente al controlador.

Visualmente en el Smartphone se debe abrir una pantalla nueva desde el "Action Button", situado en la lista de dispositivos. Así es como queda la nueva pantalla:

Como vemos, nuevamente se han reutilizado componentes, como el listado de dispositivos, o el input de fecha y hora, por lo que una vez más React nos ha ahorrado trabajo a la hora de la implementación de dicha pantalla.

Con todo esto ya solo nos queda unos últimos cambios a la organización del código de la API y la configuración pertinente en Heroku y Travis para la subida de versiones con Integración Continua.

Refactorización e Integración Continua

En este punto tenemos todo preparado. La última fase del proyecto ha sido refactorizar y organizar el código del servidor en ficheros separados según la funcionalidad. Durante todo el desarrollo se han tenido todas las funciones de base de datos y websockets junto con la lógica de los endpoints de la API y el cliente-servidor websockets.

Lo que se hizo fue refactorizar dichas funciones, que ya estaban diferenciadas dentro del propio archivo JavaScript principal, en otros archivos, para así poder usarlas a conveniencia en diferentes puntos de nuestro código.

Para ello, se han creado dos archivos diferentes, llamados "database.js" y "helpers.js", que contienen las funciones de bases de datos y algunas funciones auxiliares respectivamente. Además, a las funciones de bases de datos se les añadió una variable pasada por parámetros, siempre la misma. Esta es la librería knex, inicializada a nuestro gusto.

Se ha especificado la expresión anterior en base a que también se refactorizó la base de datos. Como ya se ha comentado anteriormente en las metodologías, se ha configurado la librería knex según el entorno en el que la aplicación se encuentre, pensando tanto en el testing local como en la integración continua de Travis CI. De esta forma se especificaron tres configuraciones de drivers de knex: development, testing y producción, seleccionado según condicionales concretas.

A su vez, se ha implementado la tecnología de hashing para las contraseñas de los usuarios. Se ha usado la librería npm de bcrypt, con la que se ha hasheado la contraseña en el registro de un nuevo usuario (nuevo endpoint de la API). Dicho

hashing se realiza también con un salt específico para cada usuario, por lo que contamos con bcrypt + salt, haciendo aún más difícil la labor a posibles atacantes. Para comparar una contraseña introducida en el login simplemente se le pasa dicha contraseña en claro a una función de la librería bcrypt, que compara la contraseña en claro con la almacenada en la base de datos. Por supuesto si intentamos hashear con bcrypt + salt la misma contraseña y la comparamos literalmente nos dan strings diferentes, por lo que hemos de pasar por la función de bcrypt.

Se han configurado los archivos correspondientes para la correcta selección de entorno. En el caso del testing local, se configuró el archivo package.json para poder reinicializar la base de datos de prueba antes y después de ejecutar los tests. Lo mismo con Travis. Se configuró el archivo pertinente de Travis del proyecto para hacer algo similar que con el testing local. En este caso lo que se realiza es la creación de la base de datos y estructura de testing para el entorno de ejecución de Travis. Como la build de Travis es única, no debemos realizar ninguna operación al finalizar la ejecución de la misma.

Para finalizar y así escribir unos cuantos tests más para la Integración Continua con Travis, se implementaron dos endpoints GET similares, uno para la recoger todos los eventos que hayan pasado en un controlador (encendido de luz, programaciones ocurridas, altas/bajas de dispositivos…), y uno para recoger todas las programaciones pendientes de un controlador, pudiendo ver fecha, mensaje de log y hora, además del dispositivo pertinente. A cada commit se han escrito un par de tests que verificaban el comportamiento de dichos endpoints, detectando un par de bugs en el tratado de la fecha y hora, que gracias a Travis se pudieron detectar ya que dicho error no pasaba en el entorno local de pruebas.

Comentado todo este proceso de desarrollo como tal, me gustaría matizar otra vez que este proyecto, pese a que les estoy presentando una versión del mismo, no está acabado. Se pretende desarrollar (incluso algunos puntos ya estarán en desarrollo para cuando sea presentado este proyecto), estas funcionalidades a corto plazo:

- Encendido de dispositivos de luz al detectarse movimiento con un sensor de movimiento
- Envío de temperatura y humedad del controlador a la API
- Evaluación de comandos de voz con el controlador

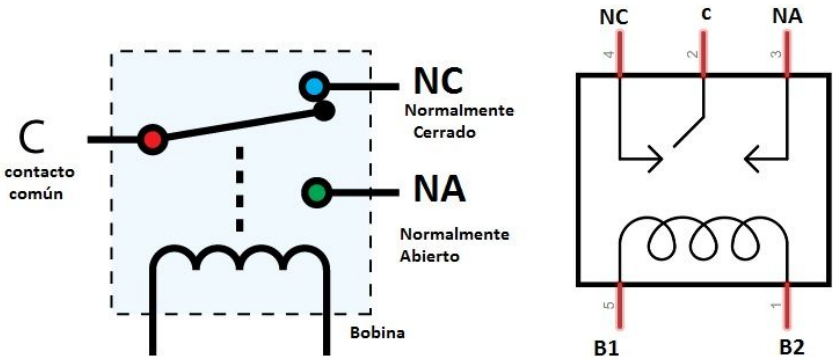
- Emisión de comandos infrarrojos a receptores infrarrojos (aire acondicionado, televisiones, etc) con un emisor IR en el controlador
- Web backend en el framework Laravel para la gestión de alta/baja de controladores nuevos
- Protección de acciones de administración (alta/baja de dispositivos) con contraseña de administrador

Resultado sobre una bombilla doméstica

Toda la arquitectura ya está montada y lista para iteraciones futuras, pero nos queda probar el sistema sobre un elemento real de una casa. Una bombilla LED de 220W de potencia puede ser un gran ejemplo.

Se ha probado con este elemento puesto que la configuración a la hora de conectarlo a un controlador Raspberry Pi es un tanto diferente. La idea no es controlar la bombilla en sí, si no controlar el relé al que va conectado.

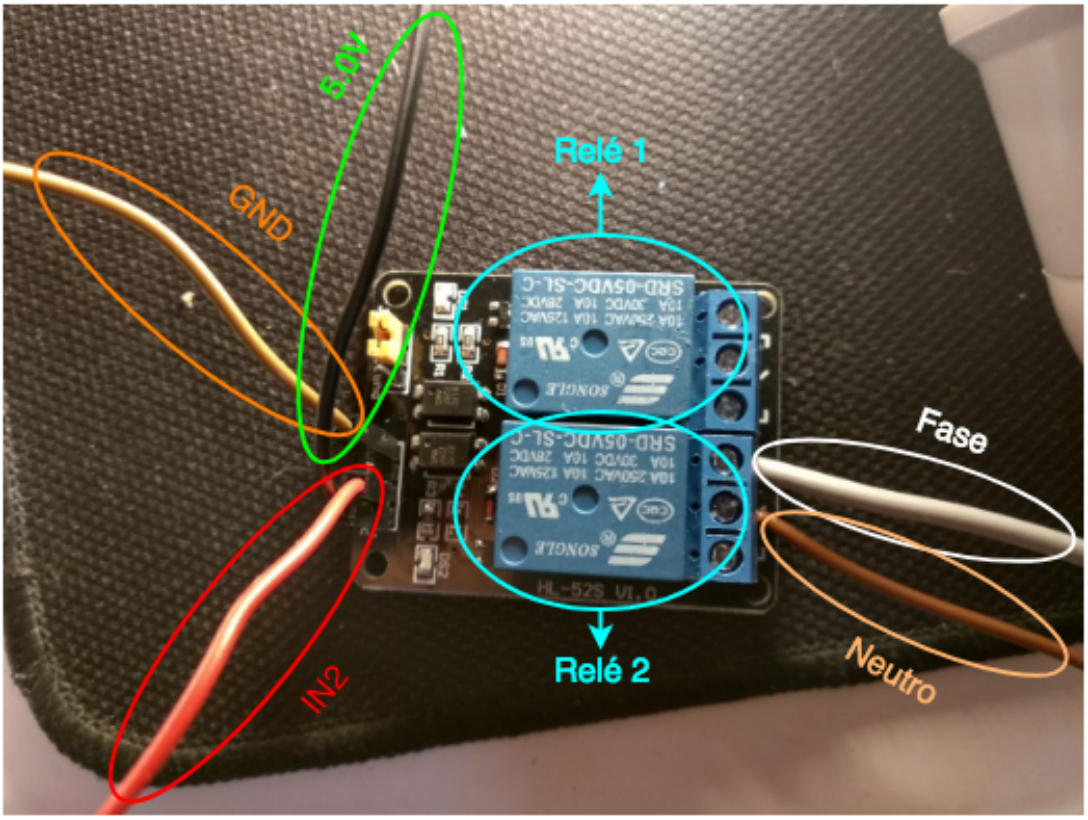
El relé es un aparato eléctrico que funciona como un interruptor, abrir y cerrar el paso de la corriente eléctrica, pero accionado eléctricamente. El relé permite abrir o cerrar contactos mediante un electroimán, tal y como vemos en el siguiente esquema:



Al meter corriente por la bobina los contactos abiertos se cierran y los cerrados se abren.

El relé que nos ocupa, sin embargo, es un tanto más especial que los utilizados normalmente en instalaciones eléctricas convencionales. Es un relé adaptado a dispositivos Arduino o Raspberry Pi, ya que cuenta con puertos especiales que ahora explicaremos.

Primero veamos la siguiente imagen del relé que utilizamos en la arquitectura presentada:

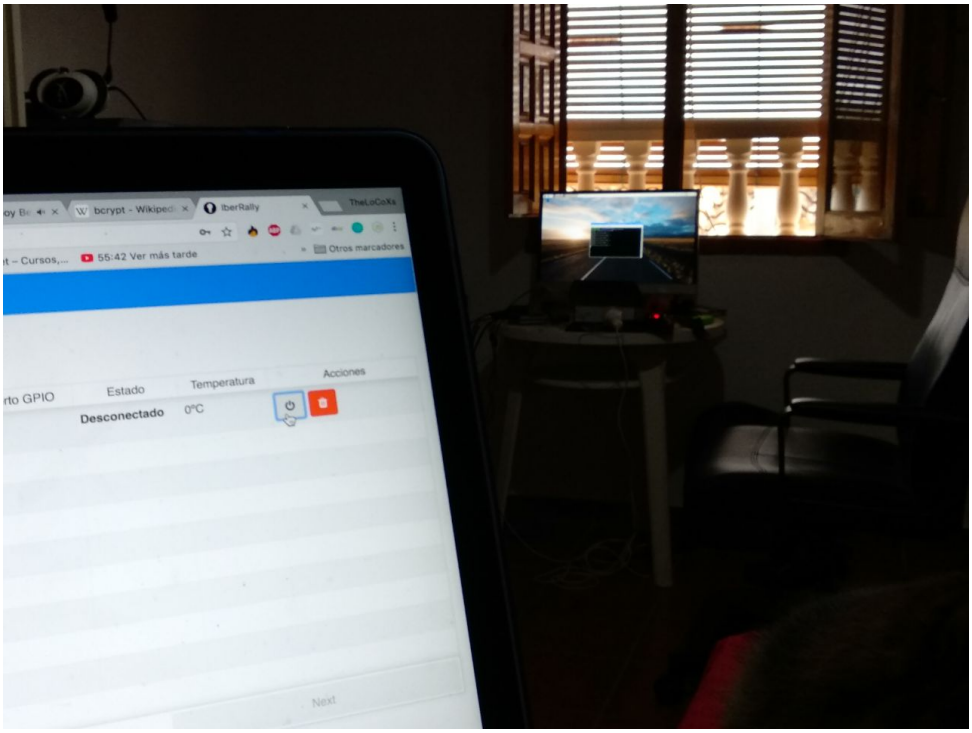


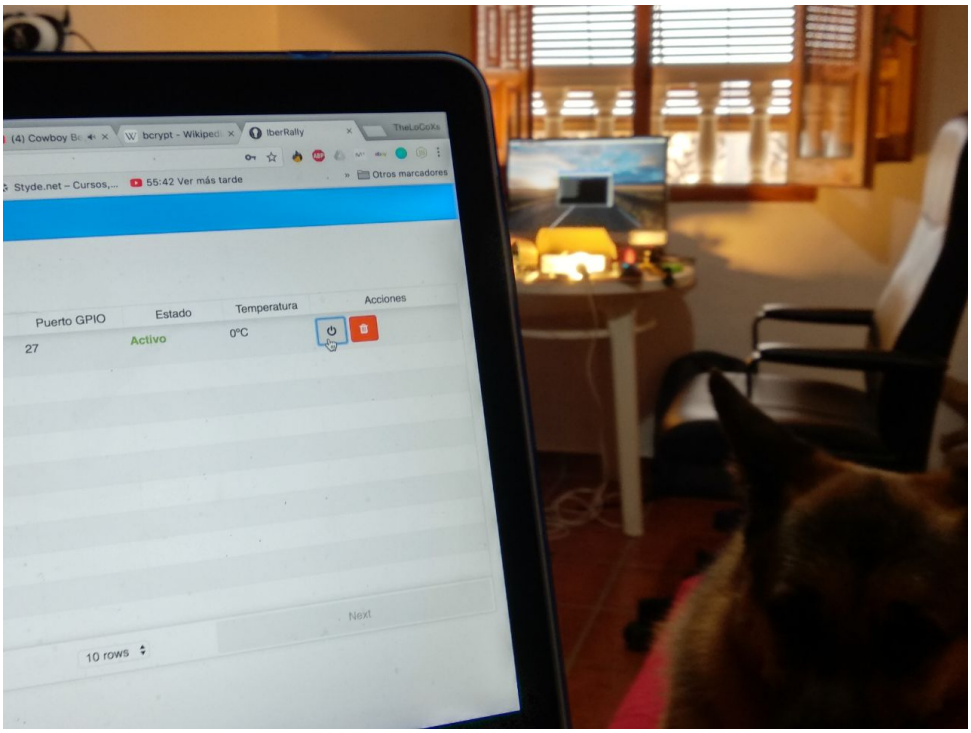
Como dijo Jack el Destripador, vayamos por partes. El dispositivo que vemos consta realmente de dos relés, tal y como está marcado en la imagen. Dichos relés están representados con dos pines macho (abajo e izquierda del dispositivo), IN1 e IN2. Para este caso únicamente utilizaremos el segundo relé, por lo que se ha conectado el cable rojo en dicho pin, que comentaremos posteriormente.

Los cables de la derecha no son más que la fase y el neutro de la bombilla LED. Pasando a los cables de la izquierda, el relé consta de cuatro pines macho para conectarlos a nuestro controlador Raspberry Pi o a una tabla de conexiones que extienda los pines de la misma. Los relés están representados como ya hemos indicado antes, con los pines IN1 e IN2. Dichos pines se han de conectar a un puerto GPIO, que es el que activa o desactiva el relé y, por tanto, es el que accionamos desde los clientes web y móvil. El cable marrón (sí, es marrón), va conectado al puerto GND del controlador, que indica toma a tierra. Por último, el cable negro va unido a un puerto de +/- 5.0V del controlador, que es el que da la potencia al relé.

En este estado ya tenemos todo lo necesario para que funcione nuestra bombilla. Lo único que debemos realizar es dar de alta un nuevo dispositivo en el cliente web, en este caso el puerto GPIO 27, indicando que el tipo de dispositivo es una bombilla.

Una vez realizado, tanto desde la web como en el móvil, podemos activar dicha bombilla a distancia, sin ser necesario estar en la localización que el dispositivo a manejar, tal y como podemos ver detrás de mi perro Dante:





BUSINESS Y FUTURO

La idea es que el servicio sea vendido a usuarios propietarios o arrendatarios de viviendas domésticas o pequeñas empresas. El abono se efectuaría mediante suscripción mensual o anual, efectuando una instalación en la vivienda mediante personal autorizado por la entidad competente. Así, evitamos grandes desembolsos iniciales a los potenciales usuarios de nuestro sistema.

Otra manera de generar ingresos es que cualquier entidad empresarial pudiera beneficiarse la arquitectura presente, es decir, ofrecerles el sistema. Para ello, se ofrecería un listado con los endpoints y su forma de uso, por ejemplo con contratos RAML. Las entidades usarían esto para crear el cliente correspondiente, mientras que la parte de servidor y controladores quedaría a cargo de nuestra arquitectura.

CONCLUSIONES

Sinceramente creo que, a nivel personal, he aprendido bastante sobre tecnologías IoT. La comunicación eficiente por websockets, la problemática de la comunicación entre servidor y controladores sin disponibilidad de IPs fijas, la gestión de los puertos BCM de una arquitectura Raspberry Pi… Son retos a los que nunca me había enfrentado en el grado, ya que mi itinerario (Ingeniería del Software) tampoco se ha centrado en este tipo de problemáticas, por lo que me ha resultado cuanto menos interesante al ver muchos proyectos caseros subidos a Internet.

También creo que he refinado bastante mi percepción sobre el usuario. Hasta este proyecto no me he dado cuenta de lo que suponía verdaderamente la interacción entre usuario-aplicación. Aunque el objetivo final no era el de la implementación de clientes para mi arquitectura, sí que es verdad que al realizarlos he intentado pensar como un usuario, y no como desarrollador, para así intentar realizar la interacción más fácil de cara a dicha entidad final. Y es que al fin y al cabo muchas veces estamos solucionando un problema para la gente, y no para nosotros.

Dicho esto, doy las gracias a los lectores por llegar hasta aquí. Espero que les haya podido entretener una décima parte de lo que me ha divertido el descubrir y desarrollar sobre estas tecnologías para mí.

REFERENCIAS Y BIBLIOGRAFÍA

1. Home Assistant: https://www.redeszone.net/2017/11/05/home-assistant-domotica-hogar/
2. Home Assistant: https://www.home-assistant.io/getting-started
3. NodeJS: https://www.w3schools.com/nodejs/nodejs_intro.asp
4. Por qué MySQL: https://www.mysql.com/why-mysql/
5. Por qué MySQL: http://www.oracle.com/us/products/mysql/mysql-wp-top10-webbased-apps-461054.pdf
6. Por qué Heroku: https://www.heroku.com/what
7. Raspberry Pi: http://histinf.blogs.upv.es/2013/12/18/raspberry-pi/
8. Raspberry Pi y puertos BCM: http://www.peatonet.com/raspberry-pi-y-los-pines-gpio-que-son-y-que-usos-practicos-les-podemos-dar/
9. Por qué Python: http://noticias.universia.es/ciencia-tecnologia/noticia/2017/07/19/1154393/sirve-phyton.html
10. Por qué React: https://www.c-sharpcorner.com/article/what-and-why-reactjs/
11. Protocolo Websockets: https://developer.mozilla.org/es/docs/WebSockets-840092-dup/Writing_WebSocket_client_applications
12. Protocolo Websockets: http://queeswebsocket.blogspot.com/2013/01/que-es-websocket.html
13. Travis CI: https://code.tutsplus.com/tutorials/travis-ci-what-why-how--net-34771
14. Por qué Git: https://git-scm.com/book/es/v1/Empezando-Acerca-del-control-de-versiones
15. Google Home: https://www.xataka.com/accesorios/google-home-venta-espana-que-puedes-que-no-puedes-hacer
16. Amazon Alexa: https://retina.elpais.com/retina/2018/01/08/innovacion/1515412624_729032.html
17. Tecnología KNX: https://www2.knx.org/

18. Explicación relé: http://www.areatecnologia.com/electricidad/rele.html